UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques
i Informàtica

GRAU DE MATEMÀTIQUES
Treball final de grau

Enumerating k-connected orientations

Author: Taras Yarema

Director: Dr. Kolja Knauer

Departament de Matemàtiques i Informàtica

Barcelona, January 24, 2021

Contents

List of Figures

Abstract

This thesis aims to study an enumeration algorithm for the k-connected orientations of a given graph, both from the theoretic perspective as well as in terms of implementation. The latter had not been done before. The process has been divided into two parts. Firstly, generating a first k-connected orientation from a $2k$-connected multi-graph, and secondly using the enumeration algorithms presented by Blind, Knauer and Valicov. Finally, for the computational part, a `SageMath` library with the implementation is presented, to generate the entire set of k-connected orientations, with a planned `SageMath` core contribution.

2010 Mathematics Subject Classification. 90C35, 05C20, 05C75, 68R10

Chapter 1

Introduction and preliminaries

Before presenting the introduction, structure and motivation of this thesis we highly recommend the reader to read this first section about complexities that will be highly used during all the present work.

If you find yourself with good knowledge of complexities you may skip directly to Section 1.2.

1.1 Some basic notions on complexities

Consider a procedure that takes X as input and outputs Y, with $n := |X|$ and $m := |Y|$. We are interested on how m grows based on n. A really simple case to illustrate this could checking if a given element is contained in an array (you may imagine it as an ordered container). This check can be naively done by going one by one, hence if the list is X with n elements the check could take up to n single elements checks. Now, this complexity notions can be denoted by $O(n)$, which is usually referred as *big O notation*. In general $O(f(n))$ is the class of functions g, such that there exists an N and a $c > 0$ such that for all $n > N$ we have $g(n) < cf(n)$. Some complexity examples could be: polynomial complexity on n (matrix multiplication can be done in $O(n^3)$), logarithmic complexity (binary search has complexity $O(\log_2(n))$) or exponential complexity (cracking a password of n characters with an alphabet of d possible characters is done in $O(d^n)$). Now, in the examples the complexity only depended on the input size n. In general, the procedure can be such that the complexity of depends on both n and m. For that we will define some additional notions.

In enumeration algorithm, we may refer to a *step* as the needed processing to move from one output to the next. The *amortized time* of a procedure can be defined as the average complexity of a procedure step based on its output. You can imagine as, even if one slow step has been made, if all the other steps are faster, then as it won't happen again for *so long* the cost is *"amortized"*. In the other hand, we denote the *delay time* as the upper bound of the complexity of every step.

We see then, that the delay and amortized times are, in fact, highly related. The amortized time is just the sum of the delays divided by the number of steps.

We say that a procedure has *polynomial amortized time* runtime if its complexity is $O(\text{Poly}(n)m)$, for some polynomial on n. *Polynomial delay* means that before the first step, between consecutive steps, and after the last step until the end, the step complexity is always $\text{Poly}(n)$ time, for some polynomial on n. As a consequence we have that polynomial delay implies polynomial amortized time.

1.2 Introduction and motivations

In an enumeration problem one receives a typically small input and wants to output every element of a typically large resulting set exactly once. In particular, we are interested in those enumerations that given an object output those object with some pre-defined property.

A simple example would be to enumerate prime numbers $2, 3, 5, 7, 11, 13, \ldots$ One –naive– approach would be to go number by number and check if the number is prime. This can be useful for some time, but soon enough problems arrive. Firstly, given your current position you have no way to know when you will find the next prime, at least thanks to Euclid's theorem you know that sooner or later you'll eventually find one. Secondly, the complexity of testing if a number is prime is well-known a hard problem. For small numbers any algorithm can be applied, but when the numbers scale, solving this problem would eventually make you the most wanted person in the world, as almost all cryptography is based on big primes.

Obviously the primes example is far from the object of study of this thesis, but it illustrates the problem in a really simple way. We want to enumerate all k-connected orientations in such a way that when we compute one, we already have a polynomial time algorithm to go to the next one, i.e. that the algorithm has polynomial time delay.

To illustrate the real problem, consider Figure 1.1 where we have an undirected graph G, formed by a set of vertices and edges [1], and an arbitrary orientation D of it. This is, picking a direction for every edge in G. It can be easily seen that this digraph D is not *strongly connected*. For example, there's no directed path from vertex 2 to vertex 1. In fact, the vertex 1 does not have any arc [2] coming into it, this is the *indegree* of it is 0. We will soon see that this property of "*how many arcs go in to (respectively out from) a vertex*" is very important to study the connectivity of the graph. Formerly, Menger [1] presented a set of important Theorems related to this properties, which will be stated in the next section of this chapter.

Now, if we would want to count all the orientations of G we would get a set of $2^{20} = 1048576$ possible orientations. Note the exponential growth of the number

[1] An edge is a *undirected* relation between two vertices.

[2] An arc is a *directed* arrow between two vertices.

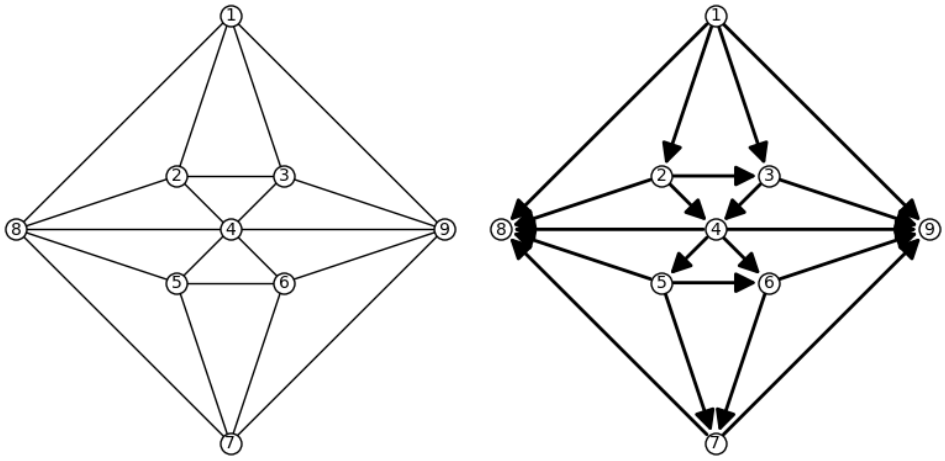

Figure 1.1: A graph G and a not-connected strong orientation D.

of orientations if we make the graph G bigger. In Figure 1.2 we have two other orientations of G. This time, though, those are *strongly connected*. One can see this if we check that for any two vertices of the graph we can find a directed path –you can imagine it as simply going through the arcs of the graph– that goes from one to the other. Both D_1 and D_2 have this property. The number of these types of orientations of G is 386190, which is around 37% of all orientations. But we can go further, and impose that we want that for any two vertices of the graph there are two different paths –different in terms that they do not share arcs– from one to the other. It's easy to see that in D_1 the vertex 1 has only on arc coming into it, hence we can deduce that it does not satisfy this property. In the other hand, D_2 does satisfy it and we say that it is 2-connected. If we count the set of 2-connected orientations of G we end up with only 1432 orientations, which is around 0.001% of all.

Note that we can further expand the presented connectivity property to high k-connectivities. Nevertheless, the presented graph G does not support orientation with higher connectivity than 2. This is related to the fact that the original G is 4-connected. In Chapter 2 an important theorem relating the connectivity of G and its orientation will be presented.

These examples illustrate the magnitude of the sets we want to enumerate. For an arbitrary undirected graph G the set of all orientation may be *really big* compared to the actual set we are interested in. This is one of the main motivation of the present thesis.

After this introductory section, rest of the chapter will be dedicated to some preliminaries notions about graph theory and classic results of the graph connectivity theory, as the Menger's theorem [1]. In the end, a first naive approach will be presented. The reason of it is to illustrate the reader on *how bad* it can get in terms of complexity when using filtering approaches in enumeration algorithms.

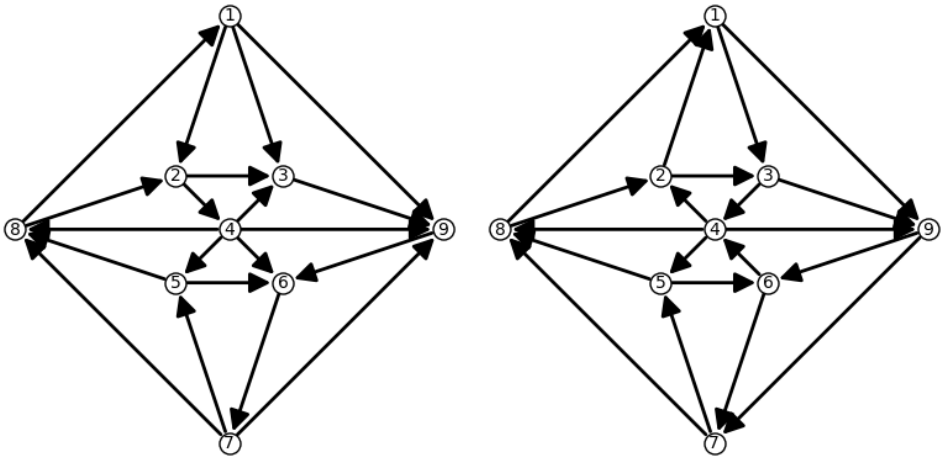

Figure 1.2: Two connected orientations of G, D_1 and D_2. The one in the right (D_2) is 2-connected.

Chapter 2 present an algorithm to generate an arbitrary k-connected orientation from a $2k$-connected graph. We will see why the initial connectivity has to be $2k$, how it affects the graph structure and presenting the crucial decomposition Theorem 2.4 by Lovasz [5]. With its approach we will analyze how to reduce a graph to the point that is *minimally connected* and study an efficient way on how to *split-off* vertices such that the graphs preserves its connectivity. Also, during all this chapter the implementation limitations will condition how the algorithms are handled, forcing us to develop some propositions and algorithms to extend the underlying algorithms.

Once the algorithm for generating one k-connected orientation is presented, we can develop in Chapter 3 the enumeration algorithms. These algorithms were presented by Sarah Blind, Kolja Knauer, and Petru Valicov in [15] (2019). The first algorithm is used to generate all orientations with a pre-defined outdegree sequence, which will be inherited from the first orientation. The second algorithm computes the following outdegree sequence. Together, can be interpreted as the *generators* of the k-connected orientations space. This space generation algorithm is the third and final algorithm: the k-connected orientation enumeration.

Finally, in Chapter 4 we will describe the technical details of the algorithm implementation. The implementation was mostly developed using open source software as `SageMath`, which is a `Python` based mathematical framework. It offers a huge amount of resources and functions to work with graphs. Since the beginning of the implementation process, we found that `SageMath` had some limitations related to our case of study, in particular with respect to multi-graphs. Even though, we are proud to say that we were able to find solutions for those limitations contributing on the enrichment of the graph related tools of `SageMath`.

1.3 Preliminaries

In this section we will introduce some basic concept about graphs and digraphs. Firstly, we will denote an undirected graph as $G = (V, E)$, where V denotes the set of vertices of the graph and E the set of edges. For directed graphs, the usual notation is $D = (V, A)$, where A is the set of arcs. All graph and digraphs will be finite. The edges in E may we written as a pair $\{u, v\}$ (or simply uv), where u, v denote two vertices of V. Note that the edges from E does not have a direction, thus both uv and vu denote the same edge. For the arcs in A we may write then as directed tuples (u, v), where u, v denote two vertices of V, and the arc goes from u to v, i.e. the order matters.

During the development of this thesis we will not consider graphs with loops, i.e. edges such that begin and end in the same vertex uu for some $u \in V$. Nevertheless, we do consider multi-edged graphs, also denoted as *multi-graphs* [3]. This means that given an edge $e = uv \in E$, resp. $a = (u, v) \in A$, there might be multiple edges $e' \in E$ such that $e' = uv$, resp. $a' \in A$ such that $a' = (u, v)$. When a graph has no multi-edges nor loops we call it *simple*.

Given an undirected graph $G = (V, E)$, we define an *orientation* of G as a directed multi-graph $D = (V, A)$ such that for every edge in E a direction is picked. This is, given $uv \in E$ we pick one of the two possible arcs $a^+ = (u, v)$, $a^- = (v, u)$.

We may encounter situations when given a digraph $D = (V, A)$ and a set of arcs $B \subseteq A$ we need to consider the graph obtained by reversing this set B. This is, replacing every $a = (u, v) \in B$ by $a^- = (v, u)$ in D. The set of reversed arcs of B is denoted as $B^- = \{a^- | a \in B\}$, and the resultant digraph of reversing the set B is noted D^B. If $B = A$, we simply note D^-.

We will also consider *mixed graphs*. These are of the form $G = (V, E \cup A)$, where E is a set of undirected edges and A is a set of directed arcs. Analogously to undirected graphs, an orientation of a mixed graph consists in fixing a direction for each of its undirected edges, i.e. the elements of E.

Given that a finite graph has a finite number of edges, the number of possible orientation are finite.

Fact 1.1. *Given a directed graph $G = (V, E)$, such that the number of edges is $m = |E|$ then there are 2^m possible orientations of G.*

Consider $G = (V, E)$ and $u \in V$. The edges of the form $uv \in E$ are called *incident* to u, or equivalently, the vertices v are called the *neighbors* of u. The set of all neighbors of u is denoted as $N_G(u)$, or simply $N(u)$, and an element $v \in N(u)$ is called *u-neighbor*. For directed graphs the definition in equivalent. Consider now two subsets $X, Y \subseteq V$, we denote $\delta(X, Y)$ as the set of edges with one endpoint in

[3] We may refer to multi-graphs simply as graphs, considering multi-graphs as the general graph object.

$X \setminus Y$ and the other endpoint in $Y \setminus X$ and the *degree* of X, Y as $d(X, Y) = |\delta(X, Y)|$. If we pick $X \subseteq V$, we denote $\delta(X) = \delta(X, V \setminus X)$ and its degree $d(X) = |\delta(X)|$.

If we have a directed graph $D = (V, A)$ and $u \in V$. The neighbors of u are the arcs of the form $(u, v), (v, u) \in E$. Note that, in the directed case, we may differentiate between those arcs that begin in u and those that end in u. So, in this case, for $X, Y \subseteq V$, we define $\delta(X, Y)$ as the arcs of A that begin in $X \setminus Y$ and end in $Y \setminus X$. We, as stated before, need to define the *outdegree* of X as $d^+(X)$, that is $|\delta(X)|$, and the *indegree* of X as $d^-(X)$, that is $|\delta(X, V \setminus X)|$.

If the set X is formed by a single vertex, this is $X = \{u\}$, we may simply note $\delta(X) = \delta(u)$ and $d(X) = d(u)$ for the undirected version, and $\delta^+(X) = \delta^+(u)$ and $d^+(X) = d^+(u)$ for the directed one. Equivalently, for the indegree $\delta^-(X) = \delta^-(u)$ and $d^-(X) = d^-(u)$.

Given a directed graph $G = (V, E)$ and two vertices $u, v \in V$, a (undirected) *path* from u to v is a sequence $P_{uv} = (e_1 = u_1v_1, \ldots, e_s = u_sv_s) \subseteq E$ such that $u_1 = u$, $v_s = v$ and $v_i = u_{i+1}$ for all $i \in \{1, \ldots, s-1\}$. Note that, as the edges in an undirected graph G does not have direction, if we have a path P_{uv} we also have the reversed path P_{vu}. We may be directly derived for directed graphs. If we consider $D = (V, A)$ and $u, v \in V$, the *directed path* between u and v is the sequence $P_{uv} = (a_1 = (u_1, v_1), \ldots, a_s = (u_s, v_s)) \subseteq A$ such that $u_1 = u$, $v_s = v$ and $v_i = u_{i+1}$ for all $i \in \{1, \ldots, s-1\}$. We say that two undirected paths from u to v, P_{uv} and P'_{uv} are *edge-disjoint* (see Figure 1.3), resp. *arc-disjoint* for directed paths, if the $P_{uv} \cap P'_{uv} = \emptyset$.

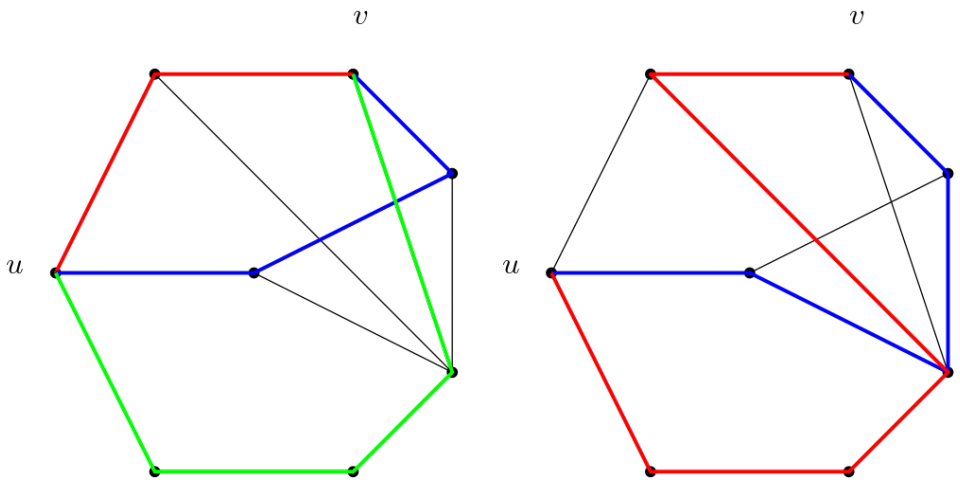

Figure 1.3: In the left, three edge-disjoint paths from u to v but only two in the right

Given $G = (V, E)$ and $u, v \in V$, the maximum number of edge-disjoint paths from u to v, resp. arc-disjoint for a directed graph D, is denoted as $\lambda(u, v)$.

Let $G = (V, E)$ be an undirected graph. We say that G is *connected* if and only if for every distinct $u, v \in V$, there's a path from u to v in E. A similar –but

yet stronger [4] – definition can be made for directed graphs. Let $D = (V, A)$ be a directed graph. We say that D is *strongly connected* –we may refer to it as *connected*– if and only if for every distinct $u, v \in V$, there's a directed path from u to v in A. We may generalize these notions for some $k \geq 1$ such that G k-connected (resp. D is k-arc-connected [5]) if and only if for every distinct $u, v \in V$, there are k edge-disjoint paths from u to v in V. (resp. k arc-disjoint directed paths from u to v in V). By Menger's Theorem, see 1.1, this is equivalent to that at least k edges (resp. arcs) have to be removed to destroy (resp. strong) connectivity.

We may sometimes refer to k-edge-connectivity of an undirected graph G. This is equivalent to the k-connectivity of G.

A *strongly connected orientation* of an undirected graph $G = (V, E)$ is an orientation $D = (V, A)$ such that it is arc-connected. More generally, a *k-connected orientation* is an orientation $D = (V, A)$ such that it is k-arc-connected.

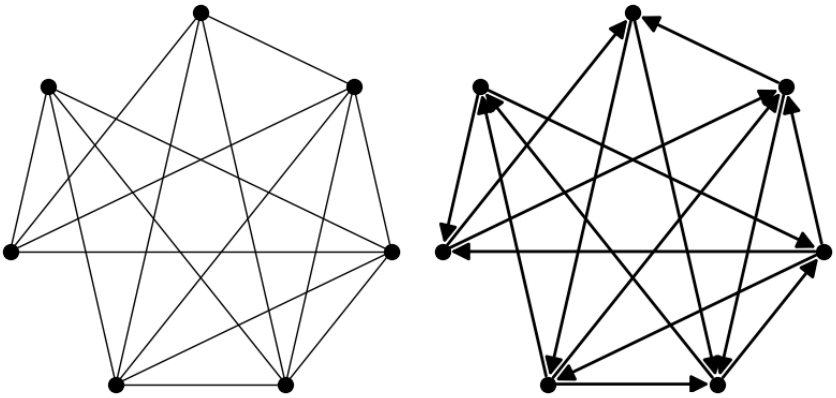

Figure 1.4: 2-connected orientation from a 4-connected undirected graph

An *edge-cut* is a subset $E' \subseteq E$ such that $G \setminus E'$ is disconnected. We say that E' *separates* $u, v \in V$ if they are in distinct components of $G \setminus E'$. This definition may be directly transfered to the set of arcs A in a directed graph $D = (V, A)$.

Now we can state Menger's theorem [1] that relates the edge-cut between two vertices and the maximum number of edge-disjoint paths between them.

Theorem 1.1 (Menger Theorem for undirected graphs)**.** *Let $G = (V, E)$ be a finite undirected graph and u and v two distinct vertices in V, then we have that $\lambda(u, v) = \min\{d(X) \mid u \in X \not\ni v\}$.*

Menger also has a directed version theorem that provides yet another characterization of k-connectivity [1].

[4] In any undirected graph a path P_{uv} will always have a reversed path which is not true for directed graphs.

[5] When there is no ambiguity, we will abbreviate k-arc-connected by saying also k-connected.

Theorem 1.2 (Menger Theorem for directed graphs). *Let $D = (V, A)$ be a finite digraph and u and v two distinct vertices in V, then we have that $\lambda(u, v) = \min\{d^+(X) \mid u \in X \not\ni v\}$.*

Note the similarity of the two theorems (1.1, 1.2). We only change the type of the graph from undirected to directed and the degree for the outdegree in the undirected version. What Menger states with these theorems is that given two vertices $u, v \in V$ the maximum number of edge-disjoint (resp. arc-disjoint) paths from u to v is equal to the minimum edge-cut that separates u and v, hence the minimality of the degrees (resp. outdegrees) of the sets $X \subseteq V, u \in X \not\ni v$.

As seen before in Figure 1.3, there are various ways of picking edge-disjoint paths from u to u. In fact, depending on which we may pick, the maximum number of edge-disjoint paths may vary. Hence, in order to determine the connectivity of a graph we can not just greedily take paths, remove them and iterate. However, in Chapter 3 we will show that it can be done efficiently in digraphs.

Finally, as a direct consequence of the directed Menger's Theorem 1.2 we have

Fact 1.2. *D is k-connected if and only if $d^+(X) \geq k$ for all $\emptyset \neq X \subsetneq V$*

1.4 A first enumeration algorithm

Given an undirected graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, a first naive approach for generating all possible orientations of it would be to go through all the possibilities of orienting the edges of G. A simple way of generating all orientations is using gray code sequences (see [11, p. 442]). Firstly we would fix a linear ordering in E, such that the elements in E could be ordered like $e_1, \ldots, e_m$. Now with an initial gray code sequence we would generate an orientation $D = (V, A)$ such that $A = A_1 \cup A_2$, where A_1 is the positive orientation of all e_i such that the i-th bit in the sequence is 1, and A_2 is the negative orientation of all e_i such that the i-th bit in the sequence is 0. We iterate over the gray code sequence and modify A accordingly. This process has runtime $O(2^m)$ and constant time delay.

Now that we generated all orientations, we need to add the process of checking on every iteration if the current orientation is k-connected, for some pre-defined $k \geq 1$. The problem with this filtering approach is that we do not have any guarantees of convergence. This is, we could go through all the possible orientations and find none that satisfies the wanted requirements. Nevertheless, we could not confirm that there are none until the end of the execution. Thus, the delay time of the algorithm is $O(2^n C(n, m, k))$ where C denotes the complexity of checking whether a k-orientation exists. In Chapter 2 we see that C is $O(m^2 + k^2 n^2 m)$ time. We can conclude that a filtering approach on enumeration algorithms is far from efficient.

The idea behind the algorithm presented in Chapter 3 is to generate the orientations in a deterministic way, that is not using filtering but once having one

orientation applying a set of operations on it such that we get a *new* orientation to continue the enumeration without repetitions.

Chapter 2

Connectivity

In this chapter we will study connectivity properties of graphs [12]. The main objective of it is to define an efficient algorithm to check whether a given undirected graph $G = (V, E)$ admits a k-connected orientation. The algorithm that we will describe is going to be constructive. That is, the check for the connectivity requirements is done while a k-connected orientation is created, thus if we can't proceed in any of the intermediate steps the given graphs does not admit a k-connected orientation. Once we have found *any* k-connected orientation of G, in Chapter 3, we will describe how to generate all such orientations from there.

We begin with a classic result from Robbins [2] characterizing the relation between edge-connectivity and strongly connected orientations.

Theorem 2.1 (Robbins)**.** *A graph G admits a strongly-connected orientation if and only if G is 2-edge-connected.*

A generalization for k-connected orientations of Theorem 2.1 is presented by Nash-Williams [3].

Theorem 2.2 (Nash-Williams)**.** *A graph G admits a k-connected orientation if and only if G is $2k$-edge-connected.*

The necessity of the Theorem directly derives from Menger's theorems (1.1 and 1.2).

Necessity of Theorem 2.2. If D is a k-connected orientation of a graph G. That is, for every $u, v \in V$ there are k arc-disjoint directed path from u to v in D. Thus, we know that both the indegree and outdegree of any $v \in V$ is at least k. And more precisely, by Menger's Theorem $d_D^+(X) \geq k$, so $d_G(X) = d_D^+(X) + d_D^+(V \setminus X) \geq 2k$ and G is $2k$-edge-connected. □

The necessity of Robbins Theorem 2.1 is in fact a particular case of the previous proof for $k = 1$. To prove the sufficiency of both theorems we will describe an important decomposition theorem by Lovasz (Theorem 2.4) which characterize $2k$-connected undirected graphs. We will prove it and use it to show sufficiency of Nash-Williams Theorem 2.2 and, as particular case, Robbins Theorem 2.1.

2.1 Generating a first k-connected orientation

Consider a graph $G = (V, E)$, $x, u, v \in V$ and a pair of edges $(x, u), (x, v)$. One can add the edge (u, v) and remove the given edge pair (see figure 2.1). This operation is noted as *splitting an edge pair*.

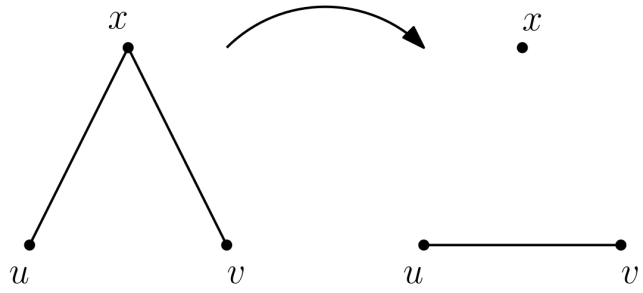

Figure 2.1: Splitting the pair $(x, u), (x, v)$

Definition 2.1 (Splitting-off)**.** *A complete* splitting-off *at x is a sequence of splitting-offs of incident edge pairs, such that x becomes isolated, and then removing x.*

Note that the splitting in Figure 2.1 is in fact a complete splitting-off at the vertex u.

The sequence of edge pairs that we split-off during a splitting-off is called a *splitting-off sequence*. In fact, given $x \in V$ the sequence will have $\frac{1}{2}d_G(x)$ splitting-offs. Note, that one of the requisites for a complete splitting-off is that the degree of x is even, that's the reason why we only study $2k$-connected graphs.

In this thesis, we want to find a splitting-off sequence that preserves connectivity. That is, if we are given an undirected graph $G = (V, E)$ and $x \in V$ such that G is k-connected, then the splitting-off of x should generate a graph G' that is also k-connected.

Consider $G = (V, E)$ such that it is k-connected, we say that a splitting-off is called *admissible* if afterwards for all $u, v \in V \setminus \{x\}$ we have $\lambda(u, v) \geq k$. Now, a complete splitting-off is called admissible if each of the $\frac{1}{2}d_G(x)$ splitting-offs are admissible. It's easy to see that not all splitting-offs may be necessarily admissible, therefore we present a result by Lovasz [5] that characterizes admissible splitting-offs.

Theorem 2.3 (Lovasz [5])**.** *Let $k \geq 1$, $G = (V, E)$ a $2k$-connected graph and $x \in V$ such that $d(x)$ is even and $|V| > 3$. Then there exists an admissible complete splitting-off at x.*

Proof. Suppose that there is no complete splitting-off at x. Then we can assume that there is no edge that can be split-off. We fix a neighbor u of x. By the assumption, for any other neighbor v of x there exists a set $U \subseteq V \setminus \{x\}$ with $d(U) \leq 2k + 1$. Denote by H the collection of all these sets. If an admissible

complete splitting-off does not exists, then these sets cover all the neighbors of x. Let $U_1, \ldots, U_l$ be the minimum number of sets of H which cover all the neighbors of x. Since

$$d(U_i \cup \{x\}) \geq 2k \geq d(U_i) - 1$$

and x has even degree, it follows that at most $d(x)/2$ edges connect x to U_i. Since x is joined to u which belongs to all U_i, it follows that two U_i's cannot contain all neighbors of x, hence $l \geq 3$.

Now consider U_1, U_2, U_3. Since U_1 could not be omitted from H, there is a neighbor of x which covered by U_1 but not by U_2 or U_3. Similarly we find that $U_2 \setminus U_3 \setminus U_1 \neq \emptyset$ and $U_3 \setminus U_1 \setminus U_2 \neq \emptyset$. Finally, we have that

$$\begin{aligned} d(U_1 \setminus U_2 \setminus U_3) + d(U_2 \setminus U_3 \setminus U_1) + d(U_3 \setminus U_1 \setminus U_2) \\ + d(U_1 \cap U_2 \cap U_3) \leq d(U_1) + d(U_2) + d(U_3) - 2. \end{aligned} \tag{2.1}$$

where all terms in the left-hand side are $\geq 2k$, but all terms on the right are $\leq 2k + 1$, hence

$$8k \leq 3(2k + 1) - 2,$$

a contradiction. □

Note that the proof of Theorem 2.3 is by contradiction hence its non-constructive. For algorithmic purposes we will later show how to efficiently find a complete splitting-off sequence at x.

The splitting-off has an inverse operation. *Pinching* a set F of edges means subdividing each edge with a new vertex and identifying the $|F|$ new vertices as a single one (see Figure 2.2). In contrast with splitting-off, pinching enough edges preserves the edge-connectivity.

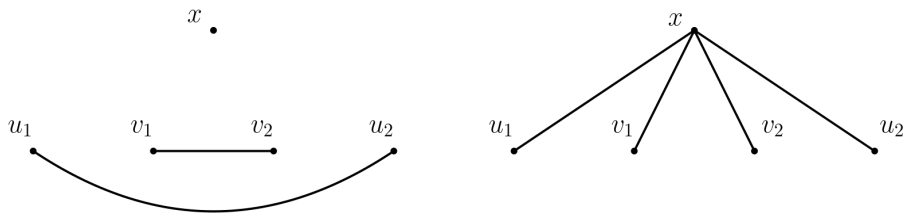

Figure 2.2: Pinching of two edges $(u_1, u_2), (v_1, v_2)$ via x

Fact 2.1. *Let $G = (V, E)$ be a $2k$-connected graph and let $F \subset E$, denote G' the graph arising from the pinching of F and denote s the new vertex in G'. Then the vertex set V is $2k$-connected in G', i.e., $\lambda(u, v) \geq 2k$ for all $u, v \in V$ Moreover G' is $2k$-connected if and only if $|F| \geq k$.*

This fact holds because the pinching does not decrease the degree of subsets of V and by Menger's theorem, G' is $2k$-connected if and only if $2|F| = d_{G'}(x) > 2k$. The pinching operation is also defined in digraphs. In a digraph, pinching a set F

of arcs consists of subdividing each arc with a new vertex and identifying the $|F|$ new vertices as a single one. As for undirected graphs, in digraphs the pinching operation preserves the arc-connectivity.

Fact 2.2. *Let $D = (V, A)$ be a k-connected digraph, let $F \in A$, denote D' the digraph arising from the pinching of F and denote s the new vertex in D'. Then the vertex set V is k-connected in D', i.e., $\lambda(u, v) \geq k$ for all $u, v \in V$ Moreover D' is k-connected if and only if $|F| \geq k$.*

Given a k-connected graph $G = (V, E)$, we say that G is *minimally k-connected* if for any $e \in E$, the graph $(V, E \setminus e)$ is not k-connected. We present an useful proposition that that guarantees a vertex of small degree in minimally k-connected graphs.

Proposition 2.1. *Every minimally k-connected graph contains a vertex of degree k.*

Proof. Firstly, given a k-connected graph $G = (V, E)$, then we know that for every $X \subset V$ it satisfies $d(X) \geq k$. We will prove this supposing that G is minimally k-connected but for all vertices $u \in V$, $d(u) > k$.

Now, suppose that G is minimally k-connected but for all $u \in V$, $d(u) > k$. Then by the undirected version of Menger's theorem we have that $\lambda(u, v) = \min\{d(X) | u \in X \not\ni v\}$, for all other vertices $v \in V$. Let's see that $\lambda(u, v) > k$ arguing over the size of $u \in X \not\ni v$. For the simple case $|X| = 1$, by hypothesis it follows $\min\{d(X)\} > k$. If $|X| > 1$, note that $d(w) > k$ for all $w \notin X$ so that all the minimum degree of X will be achieved when $X = V \setminus \{v\}$ and thus $d(X) = d(v) > k$.

In conclusion, $\lambda(u, v) = \min\{d(X) | u \in X \not\ni v\} \geq k + 1$, hence G is $(k + 1)$-connected, which contradicts the fact that is minimally k-connected. □

We introduced the notion of splitting-off and its inverse operation, also the notion of k-connected minimality. As noted in the introduction of this section, the goal is to find any k-connected orientation of a given undirected graph G. Lovasz presented a way to characterize a $2k$-connected undirected graph G such that G can be generated from an initial set of $2k$ edges connected by two vertices. The generation from this initial graph is done only by applying connectivity preserving operations: pinching a set of edges k edges or adding edges.

Formalizing the above, we have the following theorem.

Theorem 2.4 (Lovasz' decomposition [5])**.** *A graph $G = (V, E)$ is $2k$-connected if and only if it can be constructed from a pair of vertices joined by $2k$ edges by a sequence of the following operations:*

1. *add an edge between existing vertices,*
2. *pinch a set of k existing edges.*

The proof of this theorem is important because its constructive, i.e. it can be easily transformed into an algorithm.

Proof. Since adding edges and, by Fact 2.1, pinching k edges preserves $2k$-edge-connectivity, the sufficiency follows.

To prove the necessity we use induction over the number of edges $|E|$. The base case is trivial since a pair of vertices joined by $2k$ edges is clearly $2k$-connected. We have to prove that any $2k$-connected graph such that $|E| > 2k$ is obtained from a $2k$-connected graph by the operation 1 or 2. Firstly, if there exists an edge e such that $(V, E \setminus e)$ is $2k$-connected then this is done. This is, if G is not minimally $2k$-connected. Now, we may assume that G is minimally $2k$-edge-connected. Then, by Fact 2.1 and Theorem 2.3, there exist a vertex s of degree $2k$ in G and an admissible complete splitting-off at s. We denote G' and F the graph and the edges resulting from this operation, respectively. So G is obtained from the $2k$-edge-connected graph G' by pinching the k edges of F. □

Before describing the algorithm to generate a k-connected orientation of a $2k$-connected undirected graph, we prove the promised Theorem 2.2.

Proof. Suppose that $G = (V, E)$ has a k-connected orientation D. Then, for any non-empty $X \in V, d_G(X) = \rho_D(X) + \delta_D(X) \geq k + k = 2k$. Hence, by Menger's Theorem 1.1, G is $2k$-connected.

We prove by induction on the number of edges that every $2k$-connected graph has a k-connected orientation. If $|E| = 2k$ then the graph is a pair of vertices joined by $2k$ edges and orienting half of the edges is one way and the other half the other way results in a k-connected digraph. Let $G = (V, E)$ be a $2k$-connected graph such that $|E| > 2k$. By Theorem 2.4, G is obtained from a smaller $2k$-connected graph G' by operation 1 or 2. By induction G' admits a k-connected orientation D'. If G is obtained from G' by adding an edge e then the orientation of G resulting from D' by giving e an arbitrary orientation is k-connected. If G is obtained from G' by pinching a set F of k-edge then, by Fact 2.2, the orientation of G obtained from D' by pinching the set of arcs corresponding to F is k-connected. □

Based on the constructive proof of Theorem 2.4 we have a decomposition algorithm for k-connected graphs.

Algorithm 1 Lovasz' decomposition

```
function LovaszDecomposition(G = (V, E), 2k)
    while |E| > 2 do
        for e ∈ E do
            if the connectivity of G = (V, E \ {e}) ≥ 2k then
                Remove e from E
        Pick x ∈ V such that d(x) = 2k
        Compute a complete splitting-off at x
    return G = (V, E), such that V = {u, v} and |E| = 2k.
```

After applying Algorithm 1 we have an undirected graph $G' = (\{u, v\}, E')$ such that E is a set of $2k$ edges. Transforming G' into a k-connected digraph $D' = (V', F')$ is trivially done by dividing E' into two disjoint sets E'_+, E'_- and transforming those edges into arcs as seen in Figure 2.3.

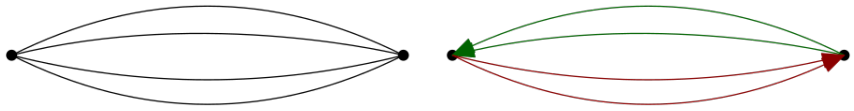

Figure 2.3: One orientation of 4 edges where the green arcs denote orienting E'_+ and the red ones orienting E'_-

Obviously, the sets E'_+, E'_- are not unique and the orientations can be reversed. The important part is that the generated orientation is k-connected. Now that we have $D' = (V = \{u, v\}, F')$ we can reconstruct it into a digraph $D = (V, F)$ such that its a k-connected orientation of G. This can be done saving every decomposition operation from Algorithm 1 (the removed edges and the split-off vertices with its neighbors) and reversing those operations with the connectivity preserving operations from Lovasz' Theorem 2.4.

Hence, we present the final algorithm to compute an arbitrary k-connected orientation of a $2k$-connected undirected graph G.

Algorithm 2 Compute k-connected orientation

function ORIENTATION($G = (V, E)$ such that its $2k$-connected)
  $G' = (V', E') \leftarrow$ LOVASZDECOMPOSITION(G, $2k$)
  Define $D = (V', F = \emptyset)$
  Make two disjoint partitions of $E' = E'_1 \cup E'_2$
  $F \leftarrow$ positive orientation of $e \in E'_+$
  $F \leftarrow$ negative orientation of $e \in E'_-$
  Reconstruct D from the removed edges and split-off vertices
  return $D = (V, F)$

Note that this algorithm do not describe how we manage the splitting-offs, or efficiently handle the minimal connectivity reduction. These steps are studied and a set of efficient algorithm are presented in the next section.

2.2 An efficient algorithm for finding a k-connected orientation

The focus of this section will be on describing an efficient algorithm to compute the complete splitting-off sequence of a vertex and another one to reduce a given graph so that its k-connected and has a vertex of degree k.

The efficient splitting-off algorithm is based on the algorithm presented by Chi Lau and Kong Yung in [13] which focuses on connectivity preservation. The results are based in Menger's and Lovasz theorems, which we already introduced in Theorems 1.1 and 1.2, and Theorem 2.3, respectively.

In the other hand, for the subgraph reduction algorithm we will use the `FOREST` procedure to find a k-connected subgraph described by Hiroshi Nagamochi and Toshihide Ibaraki at [7].

Given an undirected graph $G = (V, E)$, for any $u, v \in V$ we will denote the *local connectivity requirement* $r(u, v)$ as the edge-connectivity requirement, this is the minimum wanted connectivity between u and v. In our case, we will have *global requirement* $r(u, v)$ that is equal for all u, v. Concretely, for a fixed k we will set $r(u, v) = k$ for all $u, v \in V$. This is we impose $\lambda(u, v) \geq r(u, v) = k$ for all $u, v \in V$.

In [13] a more general version is presented, where the requirements may not be global. As stated before, we only consider a pre-defined global requirement, hence some of the results we will show are simpler. Consequentially, we may simply write $r(u, v) = k$ for all $u, v \in V$.

Before going further, we will introduce the notion of *Gomory-Hu trees*. A Gomory-Hu tree is a compact representation of all pairwise edge-connectivities of a given undirected graph. The need of this structure arises from the multi-graph problem during the implementation of the algorithm. The `SageMath` software does not support computing edge-connectivities of multi-graphs. For this reason we decided to compute it using Gomory-Hu trees. We will see how in the end of the following Section 2.2.1 presenting a general algorithm (3) that computes the connectivity of an undirected multi-graph.

2.2.1 Gomory-Hu tree construction

Let $G = (V, E)$ be an undirected graph. A *Gomory–Hu tree* of G is a weighted tree $T = (V, F, w), w : F \mapsto \mathbb{N}$ such that for every pair, $s, t \in V$, the weight of a min-cut separating them is the same in G and in T. In other words, the local edge-connectivity between $s, t \in V$ in G is equal to the weight of the minimum weighted edge on the $s - t$ path in T.

The example in Figure 2.4 the red edges in G (left figure) denote the minimum edge-cut between vertices 1 and 3. As this cut has length 2, we see that the path between 1 and 3 in T (right figure) is $(1, 2, 3)$ which has minimum weight 2, corresponding to the edge $\{1, 2\}$.

As stated before, the generic algorithms [1] used to compute Gomory-Hu trees do not consider multi-graphs. In our case, though, we do consider this type of graphs, for that we developed the previously announced subdivision algorithm to transform a multi-graph into a simple one.

[1] A Gomory-Hu tree representation can be computed using the max-flow min-cut algorithm.

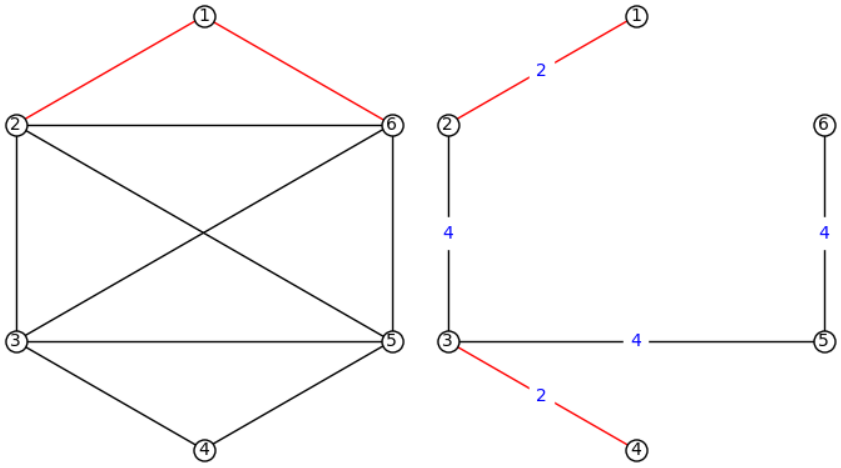

Figure 2.4: Gomory-Hu tree construction of G

Consider an undirected multi-graph $G = (V, E)$. Suppose that there exists a $l > 1$ such that $e_1, \ldots, e_l \in E$ and $e_1 = \cdots = e_l = \{u, v\}$. This is, the edge $\{u, v\}$ has *multiplicity* l. Then we can subdivide this edges via l vertices $v_1, \ldots, v_l$ such that for every $i \in \{1, \ldots l\}$ we add the edges $e_{i_1} = \{u, v_i\}$ and $e_{i_2} = \{v_i, v\}$ to E (see Figure 2.5). Now, if we remove the original e_i edges, the constructed graph G' obviously has the same connectivity as G, this is for original vertices u, v from G in G' we have that $\lambda_G(u, v) = \lambda_{G'}(u, v)$.

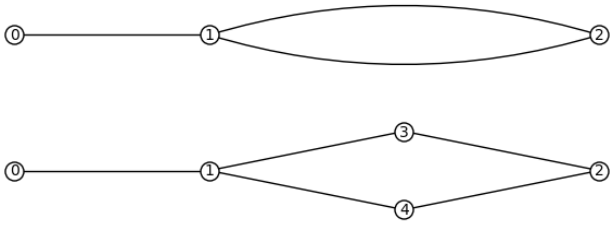

Figure 2.5: Subdivision of $e_1 = \{1, 2\}, e_2 = \{1, 2\}$ into $e_{1_1} = \{1, 3\}, e_{1_2}\{3, 2\}$ and $e_{2_1} = \{1, 4\}, e_{2_2} = \{4, 2\}$

Using this edge transformation, we can then compute the Gomory-Hu tree of the resulting graph simple graph G', T'. Note that in this transformation we create a set of new vertices $v_1, \ldots, v_l$. These vertices are contained in T', hence we need to remove them to get the wanted Gomory-Hu tree T of G.

Firstly, note that $d_G(v_i) = 2$ for all $i \in \{1, \ldots, l\}$. Without loss of generality, we pick a $v := v_i$, for some i. Hence, $w(u_i, v) \in \{1, 2\}$ for all neighbors u_i of v in T', for all $i \in \{1, \ldots, d_{T'}(v)\}$. Now, if we replace the neighbourhood by a path,

that first goes through all the neighbours $u_i \in N_{T'}(v)$, with $w(u_i, v) = 1$ and then through the neighbors with $w(u_i, v) = 2$. First, the resulting graph obviously is a tree, because we do not generate cycles. Second, if two neighbors u_i, u_j had a path through v with an edge of minimum weight $w_{i,j} \in \{1, 2\}$, then now they are connected by a path with an edge of weight $w_{i,j}$. Thus, after iterating this process for all v_i we obtain a Gomory-Hu tree T of G.

See Figure 2.6, which illustrates this reduction process from T' to T for the graph G from Figure 2.5.

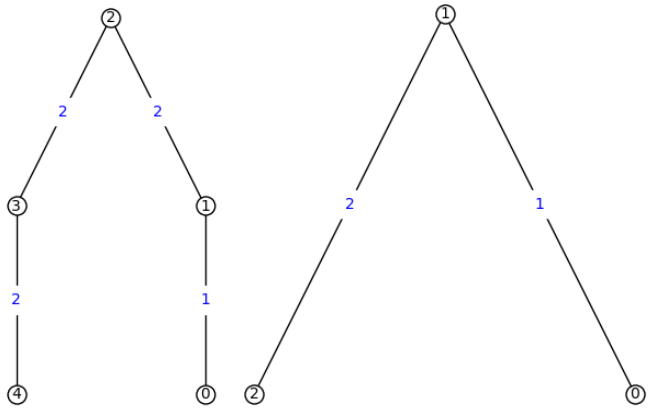

Figure 2.6: Removing vertices $v_1 = 3$ and $v_2 = 4$ from T' to get T

In Figure 2.6 we first pick vertex $v := v_1 = 3$. The two neighbors of v are $u_1 = 2$ and $u_2 = 4$ with the same weight $w_{1,2} = 2$. We join them via the path $(2, 4)$. In the second step, we pick $v := v_2 = 4$ which has only one neighbor, so we just remove v. After these two steps, we have the final Gomory-Hu tree T of G.

Now that we can compute the Gomory-Hu tree T of an undirected graph G, we present an useful result to check the local connectivity of G.

Proposition 2.2. *Consider $G = (V, E)$ and $t, u \in V$. If $k = \min_{v \in V} \lambda(u, v)$, then there is a vertex $w \in V$ such that $\lambda(t, w) = k$.*

Proof. Let $k = \lambda(u, v)$. By Menger's Theorem 1.1, there is a set $u \in X \not\ni v$ with $d(X) = k$. If $t \in X$, then $t \in X \not\ni v$ with $d(X) = k$ and $\lambda(t, v) = k$ by Menger's Theorem. If $t \notin X$, then $u \in X \not\ni t$ with $d(X) = k$ and $\lambda(u, t) = k$, by Menger's Theorem. □

From Proposition 2.2 it follows that if we pick an arbitrary vertex $t \in V$ if we compute $k = \min_{u \in V, u \neq t} \lambda(t, u)$ then G is k-connected. This minimum can be computed using the Gomory-Hu tree of G, via the minimum labels of every $u - t$ path in T, instead of computing $\lambda(t, u)$. Hence, this process is done with $O(n)$ queries instead of $O(n^2)$ (that is checking every vertex pair).

Algorithm 3 Edge connectivity of G via Gomory-Hu tree

function CONNECTIVITY($G = (V, E)$)
  Subdivide all multiple edges of G
  Compute the Gomory-Hu tree of G, $T = (V, F, w)$
  Remove all added edge during the subdivision
  Pick an arbitrary $t \in V$
  return $\min_{s \in V \setminus \{t\}}\{\text{label of the } s - t \text{ path}\}$

Hence in Algorithm 3 we presented the promised algorithm to compute the edge-connectivity of an arbitrary undirected multi-graph.

2.2.2 Efficient splitting-off

As noted before, the splitting-off of a vertex can be done in various ways. In this section we will focus on an efficient way –in terms of time complexity– to split-off to capacity a given vertex x based on [13, Theorem 3.4]. For that, let's introduce some basic notions needed to develop the efficient splitting-off algorithm. Some of the notions are similar or generalized versions of previous definitions.

Basic notions

Consider $G = (V, E)$ and a set of vertices $X \subseteq V$. Then the *requirement of the set* X, $r(X)$ is the maximum edge-connectivity requirement between every pair of $u \in X$ and $v \in V \setminus X$. We denote the *surplus* of X as $s(X) = d(X) - r(X)$. We say that X is *dangerous* if $s(X) \leq 1$ and *tight* if $s(X) = 0$. A dangerous set is said to be *maximal* if it is not a proper (neither empty nor the total) subset of any other dangerous set. Obviously, if $s(X) \geq 0$ then the connectivity requirements are satisfied. Now, consider another subset $Y \subseteq V$, then we define $\overline{d}(X, Y) = d(X \cap Y, V \setminus (X \cup Y))$.

Given an edge $e = \{u, v\} \in E$. We define the *capacity* of e as the number of copies of the multi-edge pairs $\{u, v\}$ with multiplicity l that can be split-off while satisfying the connectivity requirements. In our algorithms, we always split-off an edge pair to its capacity.

We define the *capacity* of an edge pair $e_1 = xu, e_2 = xv \in E$ to be the number of copies of the edge pair that can be split-off while satisfying the connectivity requirements for all vertex pairs other than x.

To illustrate this notion see Figure 2.7, where we have a 4-connected undirected graph G. We want to split-off to capacity the edge pair $e_1 = \{0, 1\}, e_2 = \{0, 2\}$, with $x = 0$. Note that both e_1 and e_2 have multiplicity 2. After splitting-off two copies of every edge, if we check for local connectivity between all pairs u, v such that $u, v \not/ x$ we have that $\lambda(u, v) = 4$, hence the connectivity requirements are preserved and we split-off to capacity the given edge pair. Now, if we modify the initial graph G to the one in Figure 2.8 then only e_1 has multiplicity 2, thus we

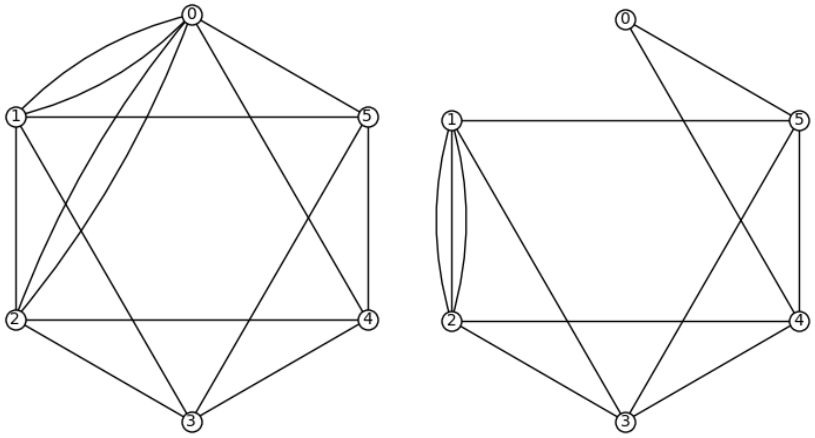

Figure 2.7: Split-off to capacity the edge pair $\{0,1\},\{0,2\}$

can only split-off one copy of the edge pair so that the connectivity requirement is preserved.

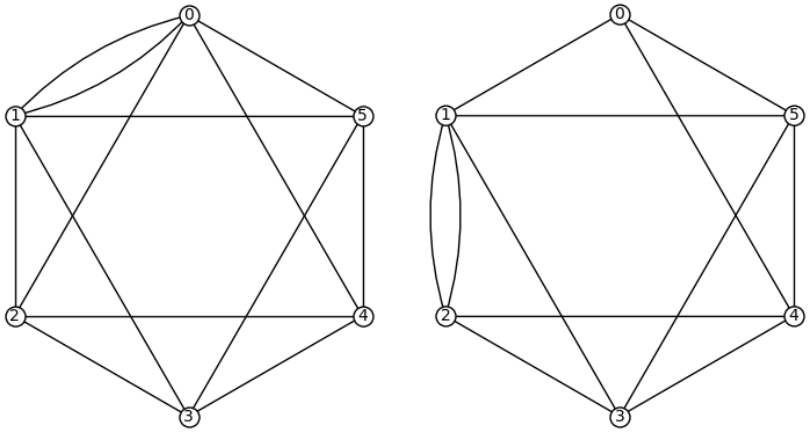

Figure 2.8: Only one copy of $\{0,1\},\{0,2\}$ can be split-off to capacity

We say that a splitting-off operation at x *voids a vertex* u if $d(x,u) = 0$ after the splitting-off.

When computing the complete splitting-off sequence we will do *splitting-off attempts*, that is, temporally splitting-off an edge pair and only committing the splitting if it preserves the connectivity requirements.

Minimizing splitting-off attempts

The focus of this section is to present Theorem 2.5 which bounds the amount of splitting-off attempts to be done during the complete splitting-off sequence

computation. For that we will need a set of results from [13].

Proposition 2.3. *A pair $\{x,u\},\{x,v\}$ is not admissible if and only if u,v are contained in a dangerous set.*

The following proposition (proved in [10]) shows that if the conditions in Mader's theorem ([13, Theorem 1.2], [4]) are satisfied, then there is no 3-dangerous-set structure as shown in Figure 2.9.

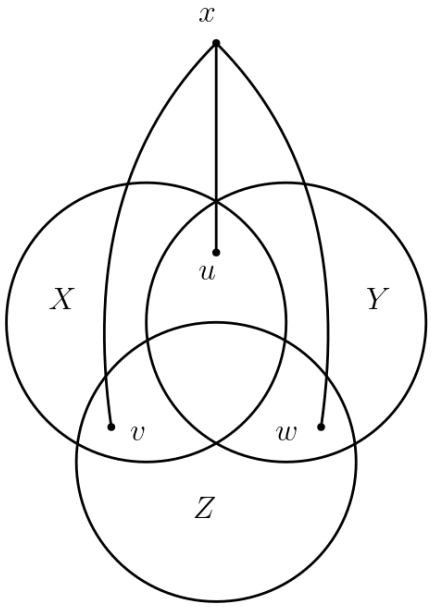

Figure 2.9: 3-dangerous-set structure

Proposition 2.4. *If $d(x) \neq 3$ and there is no cut edge incident to x, then there are no maximal dangerous sets X,Y,Z and $u,v,w \in N(x)$ with $u \in X \cap Y, v \in X \cap Z, w \in Y \cap Z$, and $u,v,w \notin X \cap Y \cap Z$.*

Proposition 2.5. *Suppose $d(x) \neq 3$. Then, for any non-admissible set $U \in N(x)$ with $|U| \geq 2$, there is a dangerous set containing U.*

Proof. We prove this by induction.

The statement holds trivially for $|U| = 2$ by Proposition 2.3. Consider $U = \{u_1, u_2, \ldots, u_{k+1}\} \in N(x)$, where every pair $\{u_i, u_j\}$ is non-admissible. By induction, since every pair $\{u_i, u_j\}$ is non-admissible, there are maximal dangerous sets X, Y such that $\{u_1, \ldots, u_{k-1}, u_k\} \in X$ and $\{u_1, \ldots, u_{k-1}, u_{k+1}\} \in Y$. Since (u_k, u_{k+1}) is non-admissible, by Proposition 2.3, there is a dangerous set Z containing u_k and u_{k+1}. If $u_{k+1} \notin X$ and $u_k \notin Y$ and there is some $u_i \notin Z$ by the maximality of X and Y, then X, Y, and Z form a 3-dangerous-set structure with $u = u_i, v = u_k, w = u_{k+1}$. By Proposition 2.4, this 3-dangerous-set structure does not exist. Hence either X, Y, or Z contains U. □

Before stating theorem 2.5 to minimize the number of split-off attempts we assume that every multiple incident edge to x, u is split to capacity. This is, removing as many admissible pairs of $\{x, u\}, \{x, u\}$ as possible.

Theorem 2.5 (Minimization of splitting-off attempts)**.** *Suppose that C is a non-admissible set and there is a vertex $u \in N(x) \setminus C$. Then, using at most three splitting-off attempts, at least one of the following operations can be applied:*

1. *splitting-off an edge pair to capacity that voids an x-neighbor,*
2. *deducing that every pair in $C \cup \{u\}$ is non-admissible and adding u to C,*
3. *contracting a tight set T containing at least two x-neighbors.*

Before proving Theorem 2.5 we will need the following proposition.

Proposition 2.6. *For $X, Y \in V$ at least one of the following inequalities holds:*

$$s(X) + s(Y) \geq s(X \cap Y) + s(X \cup Y) + 2d(X, Y) \tag{2.2}$$

$$s(X) + s(Y) \geq s(X - Y) + s(Y - X) + 2\overline{d}(X, Y) \tag{2.3}$$

Theorem 2.5. We consider three cases based on the size of C. When C is empty, we simply assign $C = \{u\}$. When $C = \{v\}$, pick the vertex v, and split off (u, v) to capacity. Either case 1 applies when either u or v becomes void, or case 2 applies in the resulting graph after $\{u, v\}$ is split off to capacity. Hence, when $|C| \leq 1$, either case 1 or case 2 applies after only one splitting-off attempt.

The interesting case is when $|C| \geq 2$ and let $v_1, v_2 \in C$. Since C is a non-admissible set, by Proposition 2.5, there is a maximal dangerous set D containing C. First, we split off $\{u, v_1\}$ and $\{u, v_2\}$ to capacity. If case 1 applies, then we are done, so we assume that none of the three x-neighbors voids, implying that $\{u, v_1\}$ and $\{u, v_2\}$ are non-admissible in the resulting graph G' after splitting-off these edge pairs to capacity. Note that the edge pair $\{v_1, v_2\}$ is also non-admissible since non-admissible edge pair in G remains non-admissible in G'. By Proposition 2.5, there exists a maximal dangerous set D' covering the non-admissible set $\{u, v_1, v_2\}$. Then the inequality 2.3 cannot hold for D and D', since that would imply

$$1 + 1 \geq s(D) + s(D') \geq s(D - D') + s(D' - D) + 2\overline{d}(D, D') \geq 2d(x, \{v_1, v_2\}) \geq 4.$$

Therefore inequality 2.2 must hold for D and D', hence

$$1 + 1 = s(D) + s(D') \geq s(D \cap D') + s(D \cup D').$$

This implies that either $D \cup D'$ is a dangerous set for which case 2 applies, since $C \cup \{u\}$ is contained in a dangerous set and hence every pair is a non-admissible pair by Proposition 2.3, or $D \cap D'$ is a tight set for which case 3 applies since v_1 and v_2 are x-neighbors.

To distinguish between case 2 and case 3, we check the existence of any tight set containing v_1 and v_2 by one splitting-off attempt of $\{x, v_1\}, \{x, v_2\}$. This is, v_1, v_2 are contained in a tight set if and only if after splitting-off one copy of $\{x, v_1\}, \{x, v_2\}$ the connectivity requirement of some pair is violated by two. Therefore, by making at most three splitting-off attempts

$$\{\{x, u\}, \{x, v_1\}\}, \{\{x, u\}, \{x, v_2\}\}, \{\{x, v_1\}, \{x, v_2\}\}$$

one of the three operations can be applied. □

Complete splitting-off sequence

Before announcing the complete splitting-off algorithm, we will describe the efficient split-off to capacity algorithm described in [13].

For a given vertex pair $\{u, v\}$, we replace $l = \min\{d(x, u), d(x, v)\}$ copies of $\{x, u\}$ and $\{x, v\}$ by l copies of $\{u, v\}$, and then determine the maximum violation of connectivity requirements using the Gomory-Hu tree based algorithm described before. Note as q the maximum violation of the connectivity requirements, then exactly $\min\{d(x, u), d(x, v)\} - \lfloor q/2 \rfloor$ copies of $\{x, u\}, \{x, v\}$ are admissible. To compute the violation of the connectivity, considering a global connectivity requirement k, we compute the difference $q = k - k'$, where k' is the connectivity of G after the first split-off to capacity.

Algorithm 4 Splitting-off to capacity

function SplitOffToCapacity($G = (V, E)$, x, k, $\{u, v\}$)
  $m \leftarrow \min\{d_G(x, u), d_G(x, v)\}$
  Remove m copies of $\{x, u\}$ and $\{x, v\}$ from E
  Add m copies of $\{u, v\}$ to E
  $q \leftarrow$ Connectivity(G)
  Add $m - \lfloor q/2 \rfloor$ copies of $\{x, u\}$ and $\{x, v\}$ from E
  Remove $m - \lfloor q/2 \rfloor$ copies of $\{u, v\}$ to E

Finally we can formalize the complete splitting-off sequence at x computation.

Algorithm 5 Efficient complete splitting-off

```
function SplitOff(G = (V, E), x, k)
    Define C = ∅
    while N(x) is not empty do
        Pick u ∈ N(x) − C
        if C = ∅ then
            Set C = {u}
        else if C = {u} then
            Pop v from C
            SplitOffToCapacity(G, x, k, {u, v})
            if d(x, u) > 0 and d(x, v) > 0 then
                Add u to C
        else
            Pop v1, v2 from C
            SplitOffToCapacity(G, x, k, {u, v1})
            SplitOffToCapacity(G, x, k, {u, v2})
            if d(x, u) > 0, d(x, v1) > 0 and d(x, v2) > 0 then
                Remove {x, v1}, {x, v2} from E
                Add {v1, v2} to E
                if Connectivity(G) < k then
                    Add {x, v1}, {x, v2} to E
                    Remove {v1, v2} from E
                    Add v1, v2 to C
```

Directly from the Algorithm 5 definition we can state that it computes the complete splitting-off sequence at x using at most $O(d(x))$ numbers of splitting-off attempts.

2.3 Complexity notes

2.3.1 Complete splitting-off algorithm

In this section we will describe the theoretic runtime of the algorithm presented in this chapter. However, we will see that some of this runtimes will have differences compared to the actual implementation presented in Chapter 4. This is because of some of this implementation use algorithm that were out of the scope of this thesis, thus we decided to simplify the implementation process. The main algorithm affected by these differences is the Gomory-Hu computation which depends on partial Gomory-Hu trees construction presented in [9].

Lemma 2.6. *Algorithm 5 computes a complete edge splitting-off sequence using at most $O(d(x))$ numbers of splitting-off attempts.*

Proof. The algorithm maintains the property that C is a non-admissible set, which holds at the beginning when $C = \emptyset$. It is clear that in case (2) the set C remains non-admissible. In case (1), by splitting-off an admissible pair, every pair of vertices in C remains non-admissible. Also, in case (3), by contracting a tight set, every pair of vertices in C remains non-admissible by Lemma 2.4.

The algorithm terminates when there is no vertex in $N(x) \setminus C$. At that time, if $C = \emptyset$, then we have found a complete splitting-off sequence; if $C = \emptyset$, then by Mader's theorem (or by the proof in section 3.1), this happens only if $d(x) = 3$ and $d(x)$ is odd at the beginning. In any case, the longest splitting-off sequence is found and the given complete edge splitting-off problem is solved.

It remains to prove that the total number of splitting-off attempts in the whole algorithm is at most $O(|N(x)|)$. To see this, we claim that each of the operations in Lemma 3.2 will be performed at most $|N(x)|$ times. Indeed, cases (1) and (3) will be applied at most $|N(x)|$ times since each application reduces the number of x-neighbors by at least one, and case (2) will be applied at most $|N(x)|$ times since each application reduces the number of x-neighbors in $N(x) \setminus C$ by one. □

Now, for the Gomory-Hu tree computation we use the $O(m)$ time algorithm in Theorem 2.6 by Nagamochi and Ibaraki [7] to construct a subgraph of G with $O(kn)$ edges. To find a complete splitting-off sequence, we can thus restrict our attention to maintaining the local edge-connectivity in this subgraph G'. This can we reduced to the following theorem.

Theorem 2.7. *There is an $O(m)$ time algorithm to construct a subgraph with $O(kn)$ edges that preserves k-connectivity, that is k-connected and has a vertex of degree k.*

Now, for the Gomory-Hu tree construction we will use the algorithm presented in [8] in addition with the fast tree packing algorithm from [9], yielding the following result.

Theorem 2.8. *A partial Gomory-Hu tree can be constructed in $O(km)$ time.*

Therefore, using Theorem 2.8, one splitting-off attempt can be performed in $O(km + n) = O(k^2 n)$ time. Note that we used $kn \leq m$ as G' is k-connected. By Lemma 2.6, the complete splitting-off problem can be solved by at most $O(n)$ splitting-off attempts. Hence we obtain the complexity of Algorithm 5 with the following result.

Theorem 2.9. *The complete edge splitting-off problem can be solved in $O(m + k^2 n^2)$ time.*

2.3.2 Orientation algorithm

This section will focus on the complexity of the whole k-connected orientation enumeration algorithm (2). This algorithm is divided in three parts: the decomposition algorithm, the arbitrary orientations of $2k$ edges and finally the recomposition algorithm. We will describe the complexities of the last parts and, in the end, the first.

For the second part, constructing an arbitrary orientation of $2k$ edges has runtime $O(k)$. For the third part, note that the original graph $G = (V, E)$ has n vertices and m edges. As we end up, after the decomposition, with 2 vertices and $2k$ edges, we removed exactly $n - 2$ vertices and $m - 2k$ edges from G. As, the operations in 2.4 have runtime $O(1)$ we will add at most $m - 2k$ edges and pinch at most $n - 2$ vertices. Thus, we can conclude that the runtime of this third part is $O(n + m)$. We have, thus, that the runtime of the second and third part of the orientation algorithm is $O(n + m)$.

There are two ways of finding a k-connected subgraph with minimum degree k. The computationally more efficient one is using Theorem 2.7 which runs in time $O(m)$. Event though, in Chapter 4 we will use a simpler to implement approach to reduce the graph to k-connected minimality. The approach is to loop over the edges of the graph and check if removing an edge preserve k-connectivity. In the end we will have a minimally k-connected graph and, by Proposition 2.1, G will have a degree k vertex, which will be split-off. The complexity of this approach will be described in 4.1 as it depends on the Gomory-Hu implementation complexity.

Theoretically, we will check for connectivity minimality of G via the $O(m)$ time algorithm in 2.7. If we reduce G to a k-connected subgraph G', by [7, Lemma 2.6] we know that G' contains a node of degree k, x. Now, before computing the complete splitting-off at x we make sure that we split-off every pair xu, xu to capacity, which has runtime $O(k^2n^2)$. Finally, by 2.9 a complete splitting-off at x can be performed in $O(m + k^2n^2)$. Now, as a splitting-off needs at least two edges, the algorithm will loop at most $m/2$ times.

Putting everything together, we can see that the runtime of the orientation construction algorithm is dominated by the splitting-off complexity. Hence, we get the following result.

Theorem 2.10. *Given a $2k$-connected undirected multi-graph G, a k-connected orientation D can be constructed in $O(m^2 + k^2n^2m)$ expected time.*

Chapter 3

Enumerating algorithms

In this chapter we will present the promised enumeration algorithm. This enumeration algorithm is based on Frank's result [6] that any pair of k-connected orientations of a graph can be transformed into each other by reversals of directed cycles and directed paths. This means that, given a k-connected orientation, we can generate following orientations applying this two operations (see Figure 3.1 for directed path reversing example of a 2-connected orientation).

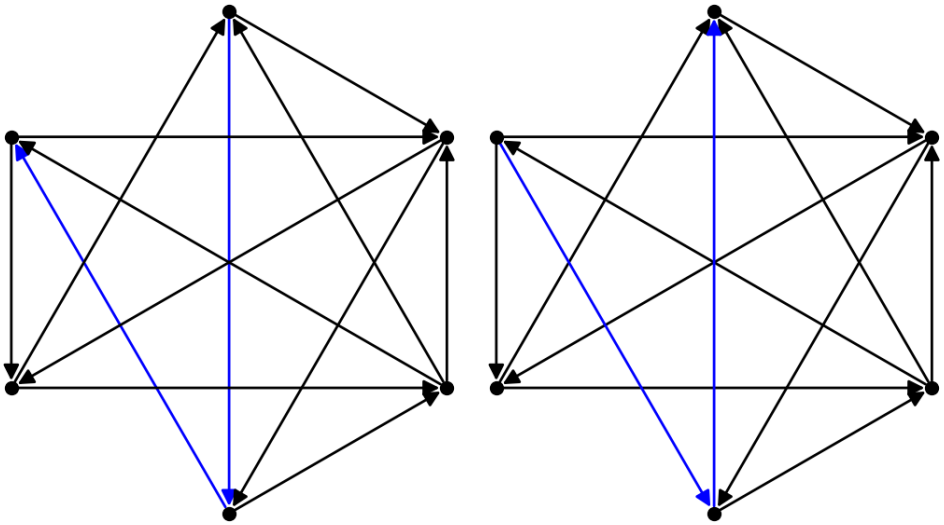

Figure 3.1: Reversing a path (blue) yields a new 2-connected orientation

3.1 Orientations with fixed outdegree sequence

Consider $G = (V, E)$ and $\alpha : V \mapsto \mathbb{N}$. We say that an orientation D of G is a *α-orientation* if $d_D^+(v) = \alpha(v), \forall v \in V$. We will denote the set $\mathcal{O}_\alpha(G)$ as the set of

all α-orientations of the graph G.

The first part of this section will be focused on presenting an algorithm for enumerating the elements of $\mathcal{O}_\alpha(G)$. But first, let's present a general result about α-orientations.

Proposition 3.1. *Let G be a graph and D and D' be two orientations of G. We have $d^+_D = d^+_{D'}$ if and only if the orientation D' can be obtained from D by reversing a set of arc-disjoint directed cycles.*

Proof. Firstly, for the sufficiency note that reversing the direction of a directed cycle does not change the outdegree function. As the cycle changes the direction of exactly two arcs of every vertex it contains, the degree does not change. Therefore, if D' is obtained from D by reversing a set of arc-disjoint directed cycles, then $d^+_D = d^+_{D'}$.

Now, for the necessity consider two orientations of G, $D = (V, A)$ and $D' = (V, A')$ such that $d^+_D = d^+_{D'}$. Consider the arcs $A \setminus A'$ of D whose direction differ in D'. Observe that in the directed subgraph $D \setminus A'$ formed by these arcs, the indegree and the outdegree coincide at each node. Thus, $D \setminus A'$ is Eulerian and can be decomposed into an arc-disjoint union of directed cycles, and D' can be obtained by reversing these set of arc-disjoint directed cycles of D. □

We give a first enumeration algorithm to construct all orientations of $\mathcal{O}_\alpha(G)$. This algorithm is given an α-orientation $D = (V, A)$ of $G = (V, E)$ and a set of arcs F, which begins empty. The output will be all α-orientations of the form $D = (V, F)$. On every iteration we pick an arbitrary arc $a = (u, v) \in A \setminus F$ and, firstly, recurse with the proper orientation $D = (V, F \cup a)$, then checking if there exists a directed path P from v to u in $D \setminus F$. If so, we reverse P as $D' = D^{P \cup a}$ and recurse with D' and $F \cup a^-$.

Algorithm 6 Backtrack search for α-orientations

```
function AlphaEnumeration(D = (V, F), F = ∅)
    if F ≠ A then
        Pick an arbitrary a = (u, v) ∈ A \ F
        AlphaEnumeration(D, F ∪ a)
        if There exists a path P from v to u in D \ F then
            AlphaEnumeration(D^{P∪a}, F ∪ a⁻)
    else
        return D
```

Let's see that Algorithm 6 uniquely generates all elements of $\mathcal{O}_\alpha(G)$.

Lemma 3.1. *Let $D = (V, A) \in \mathcal{O}_\alpha(G)$ and $F \in A$, then `AlphaEnumeration(D, F)` generates each orientation in $\mathcal{O}_\alpha(G)$ that coincides with D on F exactly once.*

Proof. We prove that each of the claimed α-orientations are generated exactly once by induction on $|A \setminus F|$.

If $|A \setminus F| = 0$, then `AlphaEnumeration(`D, F`) = AlphaEnumeration(`D, A`) =` $\{D\}$, hence we are done. Consider that $|A \setminus F| > 0$ and $a = (u, v) \in A \setminus F$. By the induction hypothesis `AlphaEnumeration(`$D, F \cup a$`)` generates each α-orientation that coincides with D on $F \cup a$ exactly once and `AlphaEnumeration(`$D^{P \cup a}, F \cup a^-$`)` generates each α-orientation that coincides with $D^{P \cup a}$ on $F \cup a^-$ exactly once, where P is a directed path from v to u. Clearly, both sets are disjoint since they differ with respect to the orientation of a. Thus, no repetitions are produced. Since (P, a) is a directed cycle by Proposition 3.1 we have $D^{P \cup a} \in \mathcal{O}_\alpha(G)$. Since $P \cap F = \emptyset$ we have that $D^{P \cup a}$ coincides with D on F.

To see that we do not miss any orientation, we prove that if there is no directed path P from u to v in $D \setminus F$, then there exists no α-orientation fixing F and reversing a. By contraposition, suppose that $D' = (V, A')$ is an α-orientation that coincides with D on F but differs on a. Then by Proposition 3.1, there is a set of arc-disjoint directed cycles in D' whose union is $A' \setminus A$. Since both digraphs coincide on F, these cycles are disjoint from F. Since both digraphs differ on a, one of the directed cycles C contains a^- in D'. Thus, the path $P = (C \setminus a^-)^-$ is a directed path in $D \setminus F$ from v to u. □

3.2 Outdegree sequences

Given an undirected graph G and a k-connected orientation D. Consider the outdegree sequence of D. We are interested in enumerating other outdegree sequences corresponding to other orientations D' (of G) that preserve k-connectivity.

In this section we will focus on constructing an algorithm to achieve this enumeration. In other words, we want to enumerate all out-degree sequences that can occur among k-connected orientations of G. Before that, we will study some needed lemmas related to outdegree sequences and connectivity.

From Proposition 3.1 the following result arises.

Lemma 3.2. *If $D, D' \in \mathcal{O}_\alpha(G)$, then D is k-connected if and only if D' is k-connected.*

Proof. Let $D, D' \in \mathcal{O}_\alpha(G)$ and D be k-connected. By Proposition 3.1 since $D' \in \mathcal{O}_\alpha(G)$ it can be obtained from D by reversing a set of edge disjoint directed cycles. But reversing a directed cycle in a digraph does not change the outdegree of the subsets of vertices of G. Therefore, after reversing the set of directed cycles to obtain D', we have $d^+_{D'}(X) \geq k$ for all non empty $X \subsetneq V$ and D' is k-connected by Menger's Theorem 1.2. □

Now, as noted in [15, Observation 1], we want to change the outdegree sequence of a k-connected orientation D such that the obtained orientation is still k-connected.

Fact 3.1. *Let D be an orientation of graph G and $X \subseteq V(G)$. If D' is obtained from D by reversing a path from a vertex u to a vertex v, then we have:*

1. $d^+_{D'}(X) = d^+_D(X)$ *if* $u, v \in X$ *or* $u, v \notin X$,

2. $d^+_{D'}(X) = d^+_D(X) + 1$ *if* $u \notin X$ *and* $v \in X$,

3. $d^+_{D'}(X) = d^+_D(X) - 1$ *if* $u \in X$ *and* $v \notin X$

In order to maintain connectivity, we have to reverse paths without decreasing the number of arc-disjoint directed paths between pairs of vertices too much.

Lemma 3.3. *[15, Lemma 8] Let P_{uv} be a directed path from vertex u to vertex v in D. For all vertices u', v', we have $\lambda_{D^{P_{uv}}}(u', v') \geq min(\lambda_D(u, v) - 1, \lambda_D(u', v'))$. Furthermore, we have $\lambda_{D^{P_{uv}}}(u, v) = \lambda_D(u, v) - 1$.* □

In a k-connected digraph D we call a directed path P *flippable* if D^P is k-connected. Lemma 3.3 implies that a path from u to v is *flippable*, if and only if any path from u to v is flippable. In this case we call the pair (u, v) *flippable*.

Consider a pair (u, v) and a directed path P_{uv}, now if we reverse this path generating the graph $D^{P_{uv}}$ we know that if we can repeat this process $k + 1$ times we will have $\lambda(u, v) > k$, hence reversing any of those paths would conserve k-connectivity. Later we will show how we will combine this procedure with a simple BFS traversal of the graph to check if a pair is flippable in $O(km)$ time.

Lemma 3.4. *[15, Lemma 10] Let $G = (V, E)$ be a graph and D, D' be two k-connected orientations of G. For every $v \in V$ with $d^+_D(v) < d^+_{D'}(v)$, there exists $u \in V$ such that $d^+_D(u) > d^+_{D'}(u)$ and (u, v) is flippable in D.* □

Lemma 3.5. *[15, Lemma 11] Let $G = (V, E)$ be a graph and D, D' be two k-connected orientations of G. For every $v \in V$ with $d^+_D(v) < d^+_{D'}(v)$, there exists $u \in V$ such that $d^+_D(u) > d^+_{D'}(u)$ and (u, v) is flippable in D.* □

Before presenting the outdegree enumeration algorithm we need some helper functions.

Algorithm 7 Helper functions for Algorithm 8

Let $D = (V, A)$ a k-connected orientation.
function IsFlippable($D = (V, A)$, $a = (u, v)$, $step = 0$)
  if $step > k$ **then**
    return True
  if Exists a directed path P_{uv} from u to v in D **then**
    return IsFlippable($D^{P_{uv}}$, a, $step + 1$)
  return False
function ReservePosOD($D = (V, A)$, F, v)
  for $u \in V \setminus F$ **do**
    if IsFlippable(D, (u, v), k) **then**
      Take a directed path P_{uv} from v to u
      ReversePosOD($D^{P_{uv}}$, F, v)
      OutdegreeEnumeration($D^{P_{uv}}$, $F \cup v$)
function ReserveNegOD($D = (V, A)$, F, v)
  for $u \in V \setminus F$ **do**
    if IsFlippable(D, (v, u), k) **then**
      Take a directed path P_{vu} from v to u
      ReverseNegOD($D^{P_{vu}}$, F, v)
      OutdegreeEnumeration($D^{P_{vu}}$, $F \cup v$)

Finally, we announce the promised outdegree sequence enumeration algorithm from [15, Algorithm 3].

Algorithm 8 Enumeration of k-connected outdegree sequences

function OutdegreeEnumeration($D = (V, A)$, $F = \emptyset$)
  if $F \neq V$ **then**
    Take an arbitrary $v \in V \setminus F$
    ReversePosOD(D, F, v)
    ReverseNegOD(D, F, v)
    OutdegreeEnumeration(D, $F \cup v$)
  else
    return d^+_D

Lemma 3.6. *Let D be a k-connected orientation of $G = (V, E)$ and $F \subseteq V$. The function `OutdegreeEnumeration(D,F)` in Algorithm 8 enumerates the k-connected outdegree sequences coinciding with D on F.*

Proof. We will show the lemma by induction on $|V \setminus F|$. If $|V \setminus F| = 0$, then `OutdegreeEnumeration(D,F)` outputs d^+_D and the claim holds. Consider now the case $|V \setminus F| > 0$ and let $v \in V \setminus F$ be the next vertex. By induction the function `OutdegreeEnumeration(D,F` $\cup \{v\}$`)` generates every k-connected outdegree sequence coinciding with D on $F \cup \{v\}$ exactly once. We have to show

that `ReversePosOD`(D, F, v) (respectively `ReverseNegOD`(D, F, v)) enumerates all k-connected outdegree sequences coinciding with D on F and having outdegree of v larger (respectively smaller) than $d^+_D(v)$. Also, we have to show that each of these outdegree sequences will be generated exactly once. Note that this implies that globally each solution is produced exactly once.

Let us prove this for `ReversePosOD`(D, F, v). So let D' be a k-connected orientation of G such that $d^+_D(F) = d^+_{D'}(F)$ and $d^+_D(v) < d^+_{D'}(v)$. By Lemma 3.4 there exists a vertex $u \in V \setminus F$ such that (u, v) is flippable, i.e., for any path P_{uv} the orientation $D^{P_{uv}}$ is k-connected, its outdegree sequence coincides with D on F and $d^+_D(v) + 1 = d^+_{D^{P_{uv}}}(v)$.

We proceed by induction on $d^+_{D'}(v) - d^+_D(v)$ to show that $d^+_{D'}$ is enumerated exactly once. So for the base case $d^+_{D'}(v) - d^+_D(v) = 1$ we have $d^+_{D'}(v) = d^+_{D^{P_{uv}}}(v)$ and by induction $d^+_{D'}$ will be enumerated exactly once by the next call of the function `OutdegreeEnumeration`$(D^{P_{uv}}, F \cup \{v\})$ and not at all by `ReversePosOD`$(D^{P_{uv}}, F, v)$ since the latter outputs degree sequences with $\alpha(v) > d^+_{D^{P_{uv}}}(v)$. Suppose now that $d^+_{D'}(v) - d^+_D(v) > 1$. We have $d^+_{D'}(v) - d^+_{D^{P_{uv}}}(v) < d^+_{D'}(v) - d^+_D(v)$, so by induction hypothesis `ReversePosOD`$(D^{P_{uv}}, F, v)$ enumerates the outdegree sequence of $d^+_{D'}(v)$ exactly once.

The analogue proof works for `ReverseNegOD` using Lemma 3.5. □

3.3 Enumeration algorithm

As stated in the beginning of this chapter, in this section we will use the algorithms 6 and 8 to present a final k-connected orientations enumeration algorithm.

Firstly, let us modify some of the helper functions defined in 7.

Algorithm 9 Helper functions for Algorithm 10

```
function ReservePos(D = (V, A), F, v)
    for u ∈ V \ F do
        if IsFlippable(D, (u, v), k) then
            Take a directed path P_uv from v to u
            ReversePos(D^{P_uv}, F, v)
            OrientationEnumeration(D^{P_uv}, F ∪ v)
function ReserveNeg(D = (V, A), F, v)
    for u ∈ V \ F do
        if IsFlippable(D, (v, u), k) then
            Take a directed path P_vu from v to u
            ReverseNeg(D^{P_vu}, F, v)
            OrientationEnumeration(D^{P_vu}, F ∪ v)
```

The idea behind this final enumeration algorithm is to join the two enumeration algorithms from the last sections so that when a $\mathcal{O}_\alpha(G)$ is totally enumerated,

a new outdegree sequence is computed, and re-iterate. Taking into account this approach, we present the final enumeration algorithm.

Algorithm 10 Enumeration of k-connected orientations

```
function OrientationEnumeration(D = (V, A), F = ∅)
    if F ≠ V then
        Take an arbitrary v ∈ V \ F
        ReversePos(D, F, v)
        ReverseNeg(D, F, v)
        OrientationEnumeration(D, F ∪ v)
    else
        return AlphaEnumeration(D, ∅)
function Enumeration(G = (V, E) such that is 2k-connected)
    Compute a k-connected orientation D = (V, A) of G
    OrientationEnumeration(D, ∅)
```

Note that we set an undirected graph G that is $2k$-connected as input. We will see in the last Chapter 4 that the check for $2k$-connectivity will be combined with the k-connected orientation computation.

3.4 Complexity notes

In [9] the overall complexity of the enumeration was bounded by using a $O(k^3n^3 + kn^2m)$ algorithm for finding an initial orientation. However, the algorithm we describe in Chapter 4 to find a first orientation has runtime in $O(kn^2m^3)$. We will discuss the resulting runtimes in the present section. For the actual implementation we have opted for a coupe of simplifications which increase the runtime slightly. We will present an analysis of the implementations in Chapter 4.

Lemma 3.7. *The function* `AlphaEnumeration(D, F)` *(6) runs with* $O(m^2)$ *time delay.*

Proof. In each recursion step the algorithm checks the presence of a directed path, which can be done by a single BFS from the source vertex u towards the target v. In general the complexity of a BFS algorithm is $O(m + n)$, however in our case the BFS tree will be constructed only on the strongly connected component of D containing u and thus the complexity is in $O(m)$. The depth of our recursion tree is bounded by m. Thus, the total time delay is bounded by $O(m^2)$. □

Theorem 3.8. *[15, Theorem 6] Let G be a graph and $\alpha : V \to \mathbb{N}$. Algorithm 6 enumerates $\mathcal{O}_\alpha(G)$ with time delay in $O(m^2)$.*

Here comes the Lemma that we promised. As we already described how we check if the pair is flippable and computing a directed path can be done in $O(m)$ time with a BFS, we have

Lemma 3.9. *Let D be k-connected, it can be decided in time $O(km)$ if (u, v) is flippable.* □

Now we can present the complexity of the outdegree enumeration algorithm stated in 8.

Lemma 3.10. *Let D be a k-connected orientation of a graph $G = (V, E)$ and $F \in V$. The function `OutdegreeEnumeration(D, F)` enumerates the k-connected outdegree sequences coinciding with D on F with time delay in $O(knm^2)$.*

Proof. We already proved the first part in 3.6. For the analysis of complexity note that in each call of `ReversePosOD` or `ReverseNegOD` for at most n times it has to be checked if a pair (u, v) is flippable. The latter can be done in time $O(km)$ by Lemma 3.9, finding a directed path from u to v is done in $O(m)$. So a call costs $O(knm)$.

Finally, the depth of the recursion tree is in $O(m)$. To see this compare the $d^+_{D'}$ of a leaf orientation with the d^+_D of orientation D at the root. Between any two calls of `OutdegreeEnumeration`, there will be a sequence of at most $deg(v)$ calls of `ReversePosOD` or `ReverseNegOD`. This way d^+_D will be approached to $d^+_{D'}$ coordinate by coordinate, where previous coordinates are not affected by modifications on latter coordinates. Thus, there are at most $\sum_{v \in V} deg(v) = 2m$ calls and we get an overall time delay of $O(knm^2)$. □

Note that Algorithm 8 has to use a separate method for finding a first k-connected orientation D of G. As noted in the beginning of this section, this preprocessing step can be done in $O(k^3n^3 + kn^2m)$. On the other hand, recall that in a k-connected orientation we have $kn \leq m$. Therefore, together with Lemma 3.10 we obtain:

Theorem 3.11. *[15, Theorem 13] Let G be a graph and $k \in N$, then Algorithm 8 enumerates all k-connected outdegree sequences of G in $O(knm^2)$ time delay.*

Lemma 3.12. *[15, Lemma 14] Let $G = (V, E)$ be a graph and α be a k-connected outdegree sequence, then $|\mathcal{O}_\alpha(G)| \geq (k-1)n + 2$.* □

Theorem 3.13. *Let G be a graph and $k \in N$. The function `Enumeration(G)` in algorithm 10 enumerates all k-connected orientations of G with $O(knm^2)$ time delay. If $k \geq 2$ the amortized time is in $O(m^2)$.*

Proof. The correctness and the delay follow directly from Theorems 3.8 and 3.11.

Let us compute the amortized time complexity as an average over the delays. Let s be the number of solutions, i.e., the total number of k-connected orientations and t be the number of k-connected outdegree sequences of G. Since by Lemma 3.12 for every k-connected outdegree sequence α there are at least $(k-1)n + 2$ orientations, we have that $t \leq s(k-1)n + 2$. Thus there exist constants c and c', such that the overall runtime of our algorithm is bounded by

$$cknm^2t + c'm^2s \leq cknm^2s(k-1)n + c'm^2s = O(m^2)s,$$

where for the last equality we use $k \geq 2$. Hence the amortized complexity is in $O(m^2)$. $\square$

Chapter 4

Sage implementation

In this chapter we will describe the actual software implementation of the algorithm presented in this thesis. We used the SageMath [17] open source mathematical software –we may refer to it as Sage–. Sage is based on Cython, which is a programming language that mixes C and Python. This composition makes it faster than other pure Python alternatives. Nevertheless, in the pure Python side we have NetworkX [16], which is a library to work with graphs and networks. We will see in the last section of this chapter, Section 4.2, that the relation between Sage and NetworkX will be useful for testing purposes.

In the first Section 4.1 the general outline of the implementation will be presented. We will describe the implemented functions and how to invoke them. Also will see the comparisons of the theoretic and final complexities of these functions. During the documentation presentation we will encounter the limitations Sage software has when working with multi-graphs.

In the final Section 4.2 we will explain how the tests of correctness were performed in addition to runtime comparison to the two main alternatives: using the naive filter approach and the Sage strong orientations iterator contributed by Kolja Knauer and Petru Valicov based on [14].

All the implementation code is hosted in a public Github repository [18] under the MIT license, which is a permissive and free license for open source software (see [20]). The author highly encourages to check the documentation offered in Github and the auto-generated documentation [19].

4.1 Library documentation

One of the most important parts of developing a software project is good documentation. This is, an explanation –should be as short and concise as possible– of the procedures (functions in our case) that compose the given library, package or service. As our library is written Sage, the author decided to write this section based on the Sage documentation (see [21] for the generic graphs documentation).

As noted before, there's an online version of the library's documentation in [19].

From now one we will suppose that the reader has a `Sage` distribution installed (versions `9.0`, `9.1` and `9.2` are supported) and the `orientations` package installed. This can be done invoking the following command from a terminal:

```
$ sage -pip install orientations
```

The implemented functions reside inside the package `orientations`. Hence when we refer to a given function `orientations.some_function` of the library, we will simply refer to it as `some_function` in this documentation. To have direct access to the package functions without the need of the package prefix, you may import all the functions from `Sage` with the following command:

```
sage: from orientations import *
```

4.1.1 `subdivide(G)`

Implements the graph subdivision algorithm. Returns an ordered tuple (G', S) where G' is the resulting graph of subdividing G on every multi-edge, and S is the of added vertices.

The complexity of this function has a worst case runtime of $O(m)$, as the graph may have only multi-edges.

`INPUT`

- `G` is an `sage.graphs.Graph`, which can be both directed or undirected.

`EXAMPLES`

Given a multi-graph $G = (V = \{0,1\}, E = \{\{0,1\},\{0,1\}\})$ then the subdivision algorithm will add two vertices $\{2,3\}$ and subdivide the two edges of E.

```
sage: G = Graph({0: [1, 1]})
sage: g, added = subdivide(g)
sage: added
[2, 3]
sage: len(added) == 2
True
sage: g.has_multiple_edges()
False
```

4.1.2 gomory_hu_tree(G)

Returns the Gomory-Hu representation of the multi-graph G using the subdivision algorithm as an underlying procedure.

The complexity depends on the native sage.graphs.gomory_hu_tree function, as after the subdivision its used to compute a intermediate Gomory-Hu tree. Finally we apply the reduction method explained in Section 2.2.1.

Now, we use the FF algorithm, that is Ford-Fulkerson, in the native Gomory-Hu construction hence we know that the complexity of this function is $O(knm)$ (see Sage Gomory-Hu implementation documentation).

INPUT

— G is an *undirected* sage.graphs.Graph.

EXAMPLES

If we consider the Petersen graph, which is an undirected graph we can compute its Gomory-Hu tree and actually get the same resulting tree as the native Sage implementation.

```
sage: g = graphs.PetersenGraph()
sage: g.edge_connectivity()
3
sage: t = gomory_hu_tree(g)
sage: t == g.gomory_hu_tree()
True
sage: g.edge_connectivity() == min(t.edge_labels())
True
```

Now if we make the graph g a multi-graph we see that we cannot compute the Gomory-Hu tree of g using the native function. Hence we use our implementation to see that the resulting graph has doubled its edge-connectivity after duplicating every edge of g. Firstly, though, we need to allow g to have multi-edges.

```
sage: g.allow_multiple_edges(True)
sage: g.add_edges(g.edges())
sage: g.edge_connectivity()
Raises error...
sage: t = gomory_hu(g)
sage: min(t.edge_labels()) == 6
True
```

4.1.3 `local_connectivity(T, u, v)`

This is a utility function, as we will work a lot with Gomory-Hu tree representations of undirected graphs G, we exported the computation of the minimum label of the $u - v$ path in T. This is $\lambda_G(u, v)$.

We use a BFS (defined in a private function `_bfs`) in T, hence the complexity is $O(m)$.

`INPUT`

- `T` is a *Gomory-Hu tree* of type `sage.graphs.Graph`.
- `u, v` are two distinct vertices of `T`.

`EXAMPLES`

As a direct example, we have that for a complete graph K_n the local edge-connectivity between all u, v is $n - 1$.

```
sage: from itertools import combinations
sage: g = graphs.CompleteGraph(7)
sage: t = g.gomory_hu_tree()
sage: all_pairs = combinations(g.vertices(), 2)
sage: all(local_connectivity(t, u, v) == 6 for u, v in all_pairs)
True
```

4.1.4 `connectivity(G)`

Returns the minimum label of the Gomory-Hu tree representation of G. This is the edge-connectivity of G.

As we use Proposition 2.2 to pre-pick an arbitrary indicator vertex the check is done in $O(n)$ time, hence this functions complexity is dominated by the underlying `Sage` implementation of the Gomory-Hu tree construction and the runtime is $O(kn^2m)$.

`INPUT`

- `G` is an *undirected* `sage.graphs.Graph`.

`EXAMPLES`

Using the Petersen graph example from before we have that.

```
sage: g = graphs.PetersenGraph()
sage: g.edge_connectivity() == connectivity(g)
```

```
True
sage: g.allow_multiple_edges(True)
sage: g.add_edges(g.edges())
sage: connectivity(g) == 6
True
```

4.1.5 `splitting_off_to_capacity(G, x, req, candidates)`

The complexity of this functions, based on Theorem 2.5 a split-off attempt can be done in $O(knm + n) = O(knm)$.

`INPUT`

- `G` is an *undirected* `sage.graphs.Graph` $G = (V, E)$.
- `x` is the vertex of V to be split-off.
- `req` denotes the connectivity requirements, hence its of the form $r = 2k$ for some positive k.
- `candidates` is the pair of x-neighbors to split-off to capacity. This parameter may be omitted and the function will pick the candidates automatically.

`EXAMPLES`

We do not illustrate the usage of this function as the usage of it is mainly done by the complete splitting-off function as an internal procedure.

4.1.6 `complete_splitting_off(G, x, req)`

Implements Algorithm 5 to efficiently compute a complete splitting-off sequence at x that preserves the global edge-connectivity requirement defined by `req`.

Modifying Theorem 2.9 with the implementation complexities we have $O(kn^2m)$.

`INPUT`

- `G` is an *undirected* `sage.graphs.Graph` $G = (V, E)$.
- `x` is the vertex of V to be split-off.
- `req` denotes the connectivity requirements, hence its of the form $r = 2k$ for some positive k.

EXAMPLES

It is well known that the Harary graph $H_{k,n}$ is a k-connected graph with n vertices. Hence let's generate a 2-connected orientation for $H_{4,7}$ (see Figure 4.1).

```
sage: G = graphs.HararyGraph(4, 7)
sage: G.plot(layout='circular').show()
Undirected Harary figure show...
sage: g = lovasz_orientation(G, 4)
sage: g.plot(layout='circular').show()
Harary orientation figure show...
sage: g.allow_multiple_edges(False)
sage: g.edge_connectivity() == 2
True
```

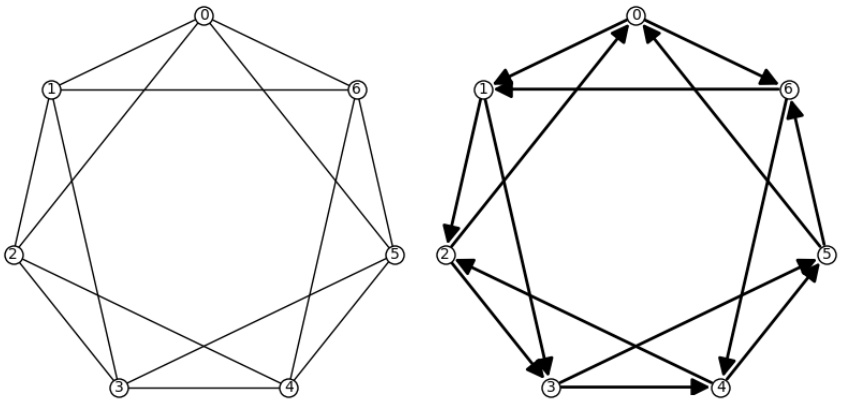

Figure 4.1: Orientation of the Harary graph $H_{4,7}$

4.1.7 `lovasz_decomposition(G, req)`

Implements Algorithm 1 for G. It returns the decomposed graph G' and a tuple of three `Python` lists that contain the added and removed edges, and the removed vertices. This is a tuple of the form `(add, rm, rm_v)`.

The function is separated in two main parts. Firstly, at every i-th step we reduces the current graph, say $G_i = (V_i, E_i)$, so that it is minimally $2k$-connected. This is done looping through the edges E_i and check if its removal violates the connectivity requirements. This check uses the `connectivity` function. After this step we get an intermediate graph $G'_i = (V'_i, E'_i)$. Secondly, we know that there exists a $2k$ degree vertex $x \in V'_i$, hence we compute the complete splitting-off sequence at x using the `complete_splitting_off`.

In the end we get a graph $G' = (\{u, v\}, E')$ where E' is a set of `req` edges between u, v.

We reduce a graph to be minimally k-connected using our connectivity function, hence on each iteration this part has runtime $O(kn^2m^2)$. Now, we already have the complexity of the complete splitting-off sequence computation, which is $O(kn^2m)$. Now, we have at most $m/2$ inner iterations, thus the whole function complexity is $O(kn^2m^3)$.

INPUT

- `G` is an *undirected* `sage.graphs.Graph`.
- `req` is the edge-connectivity requirement, that is $2k$.

EXAMPLES

For any complete graph if odd degree n we can compute its Lovasz decomposition fixing the `req` as $(n-1) = 2k$.

```
sage: G = graphs.CompleteGraph(11)
sage: g, (_, _, rm_v) = lovasz_decomposition(G.copy(), 10)
sage: len(g.vertices()) == 2
True
sage: len(g.edges()) == 10
True
sage: set(rm_v + g.vertices()) == set(G.vertices())
True
```

4.1.8 `orientation(G, k)`

Implements Algorithm 2 to compute a k-connected orientation of G.

The function uses the `lovasz_decomposition` function to compute the Lovasz's decomposition. After this it first generates a random k-connected digraph from the Lovasz's decomposition. Then it uses the `(add, rm, rm_v)` tuple to reconstruct the graph from the oriented decomposition and returns.

The whole complexity of the process is dominated by the Lovasz decomposition algorithm, as the second and last part have linear complexities in n and m. Hence this function has runtime $O(kn^2m^3)$.

INPUT

- `G` is an *undirected* `sage.graphs.Graph` $G = (V, E)$.
- `k` is the wanted connectivity for the orientation of G.

EXAMPLES

Simple usage for a complete graph K_7 and $k = 3$ generating the orientation represented in Figure 4.2.

```
sage: G = graphs.CompleteGraph(7)
sage: g = orientation(G.copy(), 3)
sage: g.is_directed()
True
sage: g.plot(layout='circular').show()
Figure show...
```

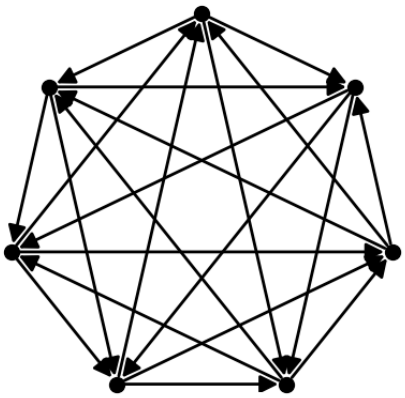

Figure 4.2: The generated 3-connected orientation of K_7

4.1.9 k_orientations_iterator(G, k)

Implements Algorithm 10 and returns an iterator to yield all the k-connected orientations of G. This function internally calls both `_alpha_orientations_iterator` and `_outdegree_sequence_iterator`, which are two internal functions that handle the computation of the α-orientations and the outdegree sequences, respectively. The implementation of these function have the same complexity as described in Section 3.4. Hence the complexity is only modified by the k-orientation computation.

We can conclude that the final implementation has *time delay* $O(knm^4)$ and *amortized time* $O(m^4)$.

INPUT

- `G` is an *undirected* `sage.graphs.Graph` $G = (V, E)$.
- `k` the connectivity of the yield orientations.

```
EXAMPLES
```

As we saw before, the Petersen graph is 3-connected, hence we can generate all its connected orientations. We can compare the resulting number of orientations with the one that the native strong orientations iterator returns. Note, though, that the strong orientations returned do not consider symmetric graphs, hence we multiply by 2.

```
sage: from sage.graphs.orientations import strong_orientations_iterator
sage: g = graphs.PetersenGraph()
sage: oris_k = len(list(k_orientations_iterator(g.copy(), 1)))
sage: oris_s = len(list(strong_orientations_iterator(g.copy()))) * 2
sage: oris_k == oris_s
True
```

4.2 Benchmarks and tests

As mentioned in the documentation section, one of the most important thing in software development is good documentations. In the other hand, a crucial point when developing is actually making sure that the developed software is doing what its suppose to do. That is, *testing* and *benchmarking*.

In our case, we already saw that there are lots of differences between the theoretic and implementations parts. First, in terms of complexity, and, secondly, in terms of implementation. We saw the limitations we faced when working with multi-graphs. This step added some additional *overhead* to the arbitrary orientation computation.

4.2.1 Tests

The tests are implemented in the source code using the `Sage` coding styles and *in-line doc tests*. Currently they are used to check if the code passes the tests when a new version of the library is pushed into Github.

In fact, the tests are highly inspired in the examples we presented in Section 4.1. For detailed reference, see the file `orientations/orientations.py` in [18].

4.2.2 Benchmarks

Here we will present some benchmarks comparing three approaches to generate k-connected orientations of a given graph. The first is the naive approach presented in the introduction chapter (`NAIVE`), the second is using the strong orientations iterator and then checking for k-connectivity (`STRONG`) and finally using the presented enumeration algorithm (`NEW`). In the `STRONG` algorithm we removed the check for connectivity when the requirement is 1, as the algorithm generated connected orientations.

The first benchmark [1] test will be done using the complete graph K_n for some low n. In Table 4.3 we have the comparison of the three algorithms when generating all strong orientations of K_3.

Algorithm	**Time**
NAIVE	0.0310
STRONG	0.0022
NEW	0.0185

Figure 4.3: Comparison of the runtimes for K_3 and $k = 1$

We see that the runtimes are fast overall, with STRONG being the fastest. Now, let's make n bigger and check for k-connectivity for some k bigger than 1, as this is the main focus of the presented algorithm in this thesis.

K_5

Algorithm	k	**Time**
NAIVE	1	0.4724
STRONG	1	0.0224
NEW	1	0.6993
NAIVE	2	0.4493
STRONG	2	0.2404
NEW	2	0.1865

K_7

Algorithm	k	**Time**
STRONG	1	96.9634
NEW	1	4475.6841
STRONG	2	1495.2407
NEW	2	963.4280
STRONG	3	1499.2306
NEW	3	6.7510

Figure 4.4: Comparison of the runtimes for K_n, $n \in \{5, 7\}$

In the Table 4.4 we have the compared runtimes for K_5 and K_7 for some higher k. Firstly, note that for K_7 we do not use NAIVE as it already gets practically impossible –in relative short time ranges– to check all orientations for a graph with 21 edges, this is $2^{21} = 2097152$ orientations. The first thing we see is that the runtime of NEW when $k = 1$ is slower than the other approaches. Even though, when $k > 1$, we see that the runtime of NEW is faster than the other algorithms.

We also implemented some benchmarks for the Harary graph $H_{k,n}$, which we already introduced in the previous section. This graph is really useful as we do not need to check if the original undirected graph is k-connected. In the other hand, we wanted to test non-complete graphs that had high connectivities.

[1] All the benchmarks were computed using SageMath 9.1 in a DigitalOcean droplet with 2 GB of RAM and 2 vCPUs.

In the first Chart 4.5 we fix $k = 2$, hence the yield orientations will be strongly connected. As we already seen in the previous benchmarks, `NEW` does not have good runtimes compared to the other.

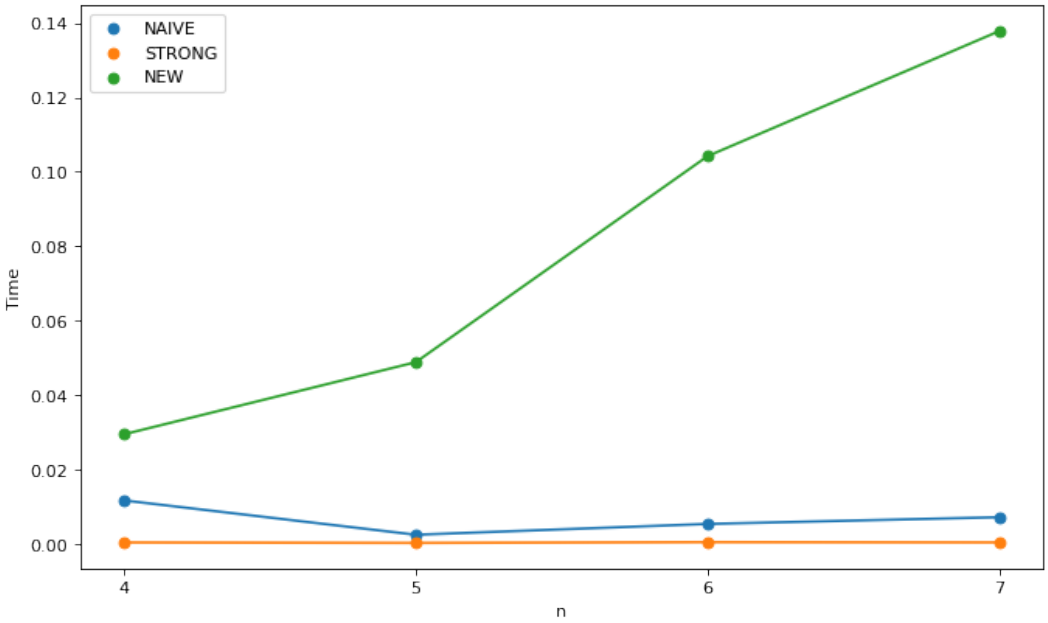

Figure 4.5: Runtime comparison for the Harary graph $H_{2,n}$

When we take $k = 4$ (see Chart 4.6), the algorithm yield 2-connected orientations. The `NEW` algorithm is the fastest, keep the runtime under one second for all n.

For $H_{6,7}$, we only ran `STRONG` and `NEW`. In this case `STRONG` took 1540 seconds (around 25 minutes), while `NEW` took only 6.

Finally in Chart 4.7 we present the `NEW` runtimes for higher k and n values for the Harary graph.

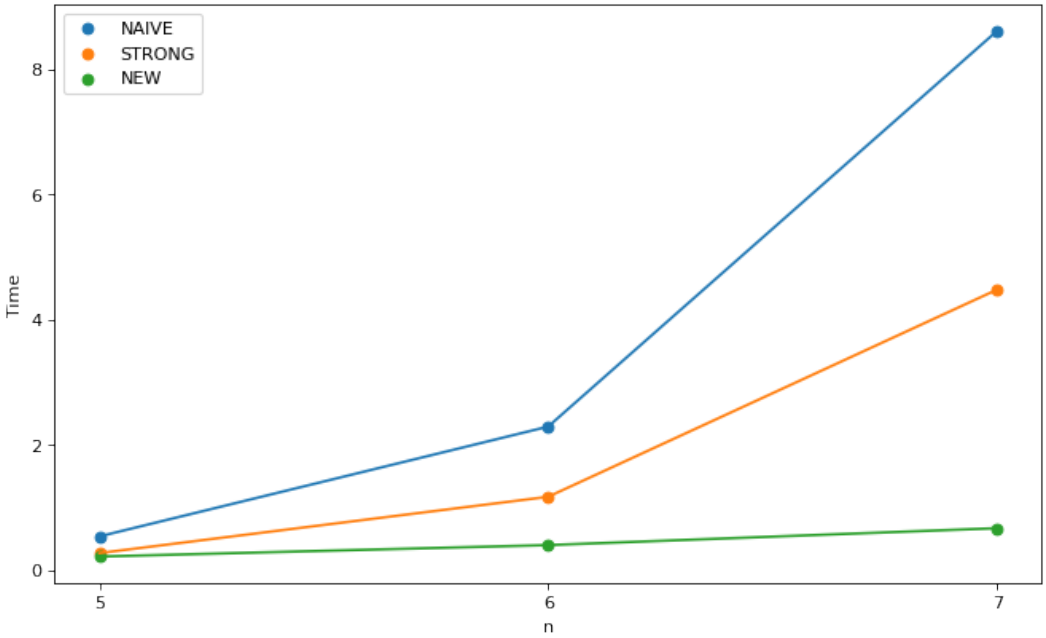

Figure 4.6: Runtime comparison for the Harary graph $H_{4,n}$

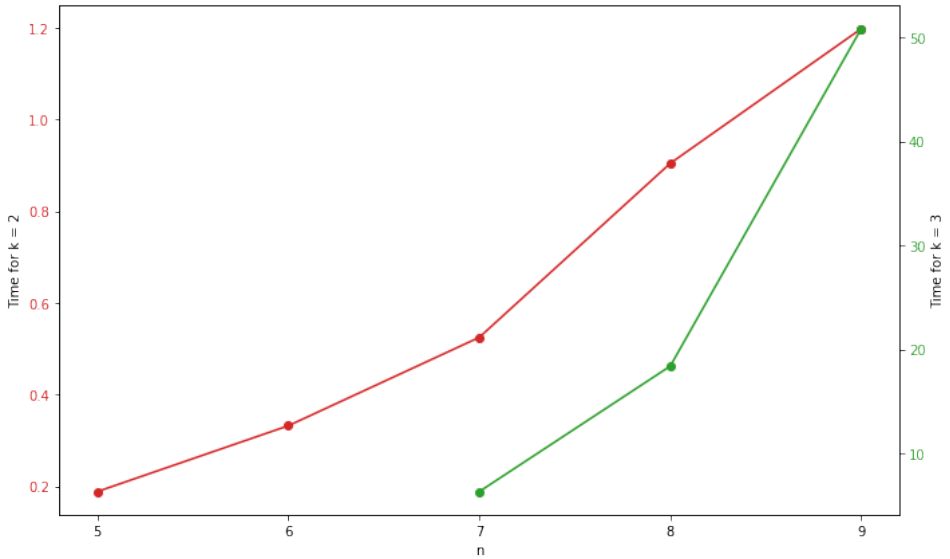

Figure 4.7: `NEW` algorithm runtime for $H_{k,n}$ in seconds

Chapter 5

Conclusions

As presented in the introduction of this thesis, a k-connected orientations enumeration algorithm has been implemented. The process was driven by a first implement, then document approach. Since the beginning of the research, the `Sage` notebook environment has been present to analyze, test and visualize whats was going on. Nevertheless, the implementation process took more than expected, influenced by the `Sage` limitation that we described in this last implementation Chapter 4. It is very satisfactory for the author, though, that the implementation of the algorithms has been successful and that, as seen in the benchmarks Section 4.2, improved the existing algorithms. Not to say that there are room for improvements of the code, for example, using the subgraph reduction algorithm from [7], which would improve the initial enumeration computation runtime, and hence the delay and amortized time of the enumeration algorithm.

In terms of future plans, a contribution to the `Sage` core is planned to be done during this year 2021. This process could be long and even the setup could take some time. Event though, thanks to the project structure of our implementation, the source files use the same structure and documentation structure as in `Sage`, the contribution process won't be that complicated.

As a final note, the author would want to thank Dr. Kolja Knauer for proposing this thesis, helping and giving good advise through the research process. I hope the present work done in some way or another comes helpful in future research.

Bibliography

[1] K. Menger. "Zur allgemeinen Kurventheorie". German. In: *Fundam. Math.* 10 (1927), pp. 96–115. ISSN: 0016-2736; 1730-6329/e.

[2] H. E. Robbins. "A theorem on graphs, with an application to a problem of traffic control". English. In: *Am. Math. Mon.* 46 (1939), pp. 281–283. ISSN: 0002-9890.

[3] C. St. J. A. Nash-Williams. "On orientations, connectivity and odd-vertex-pairings in finite graphs". English. In: *Can. J. Math.* 12 (1960), pp. 555–567. ISSN: 0008-414X; 1496-4279/e.

[4] W. Mader. "A reduction method for edge-connectivity in graphs". English. In: *Advances in graph theory*. 1978, pp. 145–164.

[5] László Lovász. *Combinatorial problems and exercises*. English. Amsterdam-New York-Oxford: North-Holland Publishing Company. Budapest: Akademiai Kiado, Publishing House of the Hungarian Academy of Sciences. 551 p. hbk: Dfl. 95.00; $ 46.25; pbk: Dfl. 55.00; $ 26.75 (1979). 1979.

[6] Andras Frank. "A note on k-strongly connected orientations of an undirected graph". English. In: *Discrete Math.* 39 (1982), pp. 103–104. ISSN: 0012-365X.

[7] Hiroshi Nagamochi and Toshihide Ibaraki. "A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph". English. In: *Algorithmica* 7.5-6 (1992), pp. 583–596. ISSN: 0178-4617; 1432-0541/e.

[8] Ramesh Hariharan, Telikepalli Kavitha, and Debmalya Panigrahi. "Efficient algorithms for computing all low s-t edge connectivities and related problems". English. In: *Proceedings of the eighteenth annual ACM-SIAM symposium on discrete algorithms, SODA 2007, New Orleans, LA, USA, January 7–9, 2007*. New York, NY: Association for Computing Machinery (ACM); Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 2007, pp. 127–136. ISBN: 978-0-89871-624-5.

[9] Ramesh Hariharan et al. "An $\widetilde{O}(mn)$ Gomory-Hu tree construction algorithm for unweighted graphs". English. In: *Proceedings of the 39th annual ACM symposium on theory of computing, STOC 2007. San Diego, CA, USA, June 11–13, 2007*. New York, NY: Association for Computing Machinery (ACM), 2007, pp. 605–614. ISBN: 978-1-59593-631-8.

[10] Yuk Hei Chan et al. "Degree bounded network design with metric costs". English. In: *SIAM J. Comput.* 40.4 (2011), pp. 953–980. ISSN: 0097-5397; 1095-7111/e.

[11] Donald E. Knuth. *The art of computer programming. Volume 4A. Combinatorial algorithms. Part 1*. English. Upper Saddle River, NJ: Addison-Wesley, 2011, pp. xv + 883. ISBN: 978-0-201-03804-0/pbk.

[12] Olivier Durand de Gevigney. "Graphs Orientations : structures and algorithms". Theses. Université de Grenoble, Oct. 2013. URL: https://tel.archives-ouvertes.fr/tel-00989808.

[13] Lap Chi Lau and Chun Kong Yung. "Efficient edge splitting-off algorithms maintaining all-pairs edge-connectivities". English. In: *SIAM J. Comput.* 42.3 (2013), pp. 1185–1200. ISSN: 0097-5397; 1095-7111/e.

[14] Alessio Conte et al. "Directing road networks by listing strong orientations". English. In: *Combinatorial algorithms. 27th international workshop, IWOCA 2016, Helsinki, Finland, August 17–19, 2016. Proceedings*. Cham: Springer, 2016, pp. 83–95. ISBN: 978-3-319-44542-7/pbk; 978-3-319-44543-4/ebook.

[15] Sarah Blind, Kolja Knauer, and Petru Valicov. "Enumerating k-Arc-connected orientations". English. In: *Algorithmica* 82.12 (2020), pp. 3588–3603. ISSN: 0178-4617; 1432-0541/e.

[16] *NetworkX*. 2020. URL: https://networkx.org/.

[17] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.1.0)*. 2020. URL: https://www.sagemath.org.

[18] 2pac. *tarasyarema/orientations: Orientations package release*. Version v0.3.2. Jan. 2021. DOI: 10.5281/zenodo.4459718. URL: https://doi.org/10.5281/zenodo.4459718.

[19] Taras Yarema. *Orientations reference*. 2021. URL: https://orientations.taras.cc.

[20] *The MIT License*. 2021. URL: https://mit-license.org/.

[21] The Sage Developers. *Sage Reference Manual: Graph Theory, Generic graphs (common to directed/undirected)*. 2021. URL: https://doc.sagemath.org/html/en/reference/graphs/sage/graphs/generic_graph.html.